DEPARTMENT OF DEFENCE

DSTO

Information Technology Division

RESEARCH NOTE
ERL-0715-RN

AN INDEPENDENT EVALUATION OF THE DECLARATIVE ADA DIALECT

by

Gina Kingston and Stephen Crawley

ELECTRONICS RESEARCH LABORATORY

AR-008-466

DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

RESEARCH NOTE
ERL-0715-RN

AN INDEPENDENT EVALUATION OF THE DECLARATIVE ADA DIALECT

by

Gina Kingston and Stephen Crawley

SUMMARY

This paper provides an independent evaluation of the Declarative Ada Dialect (DAD), which allows functional-style programming in Ada and was developed by the University of Queensland under a Research Agreement with DSTO. It describes the use of DAD and discusses its benefits and limitations.

JAN 94

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory, PO Box 1500, Salisbury, South Australia, 5108.

CONTENTS

Page No.

LIST OF FIGURES

Page No.

1 Introduction

Ada is a procedural programming language developed in the 1980's under the direction of the US Department of Defense [1]. One of the reasons for its development was to reduce the number of languages supported by the Department. Most of the operational systems being developed for the Australian Department of Defence are programmed in Ada.

Since Ada's development, functional programming has become popular for a variety of tasks — including prototype programming during the early stages of the software lifecycle. Ada does not support rapid prototyping, so a second language is required for prototyping. This implies additional overheads including the provision of additional programming language tools, support and training. As an alternative, Dr Paul Bailes of the Department of Computer Science at the University of Queensland proposed that Ada should be extended to support the functional paradigm and suggested that this could enable rapid prototyping to be performed in Ada.

This proposal led to a research agreement (DST 89/8809) between DSTO's Information Technology Division and the University of Queensland. The agreement consisted of two main components. The first called for the design of functional extensions to the Ada language, which later became known as the Declarative Ada Dialect (DAD). The second component called for the development of a preprocessor (written in Ada) to convert DAD source code into standard Ada. This work was done by the University's Key Centre for Software Technology over a three year period.

During this time, a number of papers were delivered to DSTO, giving sample programs written in DAD (and its predecessors) and describing the syntax, semantics and implementation of the Declarative Ada Dialect. On completion of the agreement the following were delivered:

1. A final report "DAD — an Ada Preprocessor for Functional Programming" [2]. This is a covering document for 14 separate papers, 8 of which are bound with the report; the remaining 6 are previously published technical reports.
2. The source code for the DAD preprocessor and the DAD libraries.
3. The source code for several sample DAD programs.

A previous ITD paper [3] provides a reading guide to the documents delivered under the agreement.

This paper provides an independent evaluation of DAD assessing its usability and maturity. Section 2 provides an introduction to the DAD language. Section 3 describes a case study which was implemented as part of the evaluation. Section 4 discusses the problems encountered while compiling and executing the delivered DAD programs (details of which are given in Appendix E.C.2) and the case study. Section 5 analyses some performance characteristics of the DAD preprocessor and its implementation. Section 6 discusses some aspects of the usability of DAD in general, and as a prototyping language. Finally Section 7 offers some advice about the use of DAD.

2 An Overview of the DAD Language

The DAD language was developed to provide Ada programmers with access to the functional programming paradigm for the development of rapid prototypes which could be incrementally refined to efficient, robust, Ada programs.

The language was developed as an extension of the Ada language which can be converted into pure Ada by a preprocessor written in standard Ada. The following quote from [4] justifies the choice of a preprocessor implementation:

> "A preprocessor implementation, translating our Ada extensions into "raw" Ada, guarantees: that prototype components will in the final analysis be written in precisely the same language as production components; that provided the preprocessor is also written in Ada, the prototyping language will be portable to any Ada platform; that details of code generation and error analysis can be kept to a minimum."

The DAD syntactic extensions are described in the DAD Language Reference Manual [5], which should be read in conjunction with the Ada LRM [1]. The most important additions are first-class functions and

lazy types which will be described below. The reader is refered to the DAD LRM for other syntactic extensions.

The current preprocessor generates code which uses generic packages containing appropriate tasks for the implementation of first-class functions and lazy types. The primary and optimised translations to standard Ada [6, 7], and details of how these translations were developed [4, 8, 9], are provided in the delivered documents.

To adhere to Ada's strong typing rules, first-class functions must be typed. A first-class function type declaration has the form:

```
type identifier is function [formal_part] return type_mark.
```

For example, the following declaration could be used to declare a function type which took two integer arguments and returned a boolean:

```
type int_comparison is function (i,j:integer) return boolean.
```

A particular first class function is specificed using the following format

```
function identifier:fn_type_mark [[formal_part] return typemark].
```

A comparison function for integers, `less_than` can be specified in either of the following forms

```
function less_than:int_comparison
```

```
function less_than:int_comparison (greater,lesser:integer) return
   boolean.
```

The first line shows the simplest specification possible; the second example shows how meaningful identifiers can be used instead of the defaults provided in the type declaration. Note that, while the identifiers may be renamed, the types must remain the same.

Lazy types are simply identified by the key word "lazy" in their type declarations:

```
type indentifier [discriminant] is lazy type_definition
```

```
type lazy_boolean is lazy boolean.
```

The following is an extract from the `mapprime` program which was supplied by the DAD developers and is documented in [10]. The procedure shows how the DAD library `stream_pack` which implements lazy streams can be used in declaring the `map` function, and gives a example of its use to display the squares of prime numbers. The function `square` is passed as an argument to the `map` function, and therefore must be first-class. The result from the `map` function is also first-class and is of type `is_to_is`.

```
package int_stream is new stream_pack (integer); use int_stream;

function up_to (start,finish: integer) return stream;
function sieve (seed: in stream) return stream;

type int_to_int is function (k: integer) return integer;
type is_to_is is function (l: stream) return stream;

function up_to (start, finish: integer) return stream is
...
end up_to;

function sieve (seed : in stream) return stream is
...
end sieve;

function square : int_to_int (k: integer) is
begin
```

```
  return k * k;
 end square;

 function map (f: int_to_int ) return is_to_is
  function : is_to_is is
  begin
   if is_null_stream (l) then
    return l;
   else
    return
     f (head (l))
     &
     map (f) (tail (l));
   end if;
  end;
begin
 declare
  mapsq : is_to_is := map (square);
  ints : stream := up_to (2,1000);
  primes : stream := sieve (ints);
  psq : stream := mapsq (1 & primes);
  i:integer:=0;
 begin
  while not is_null_stream (psq) loop
   i := i+1;
   put (i); put (head (psq)); new_line;
   psq := tail (psq);
  end loop;
 end;
```

Most of the other extensions are mainly syntactic sugar. For example, the term **dynamic** is used declare recursive records. These are implemented using access types which point to the records.

3 Case Study

The following case study was performed by the authors to determine how easy it is for an inexperienced DAD programmer to create and debug DAD programs. This section describes the development strategy for the case study. Implementation problems, performance issues and conclusions on DAD's usability arising from this study and the DAD programs provided are discussed in Sections 4, 5 and 6 respectively.

3.1 Problem Statement

The case study is based on the example given in the following news item from `comp.lang.ada`, posted on 29 Jan 90 with the subject: "Teaching concurrency in Ada".

> For a tasking example that can be developed from a very simple one to include most of the task syntax, I use a remote temperature sensor problem originally introduced by Young in "Real-time languages".
>
> In the basic form you have a set of tasks each sampling the temperature of a 'furnace' at certain intervals. The tasks all use the same thermometer, so they need to be synchronized by a THERMOMETER tasks. They also share the line to a mainframe on which they send data packets. This requires an OUTPUT task.

In a later step, the sensor is made to receive messages from the mainframe that change the sampling intervals. For this, an INPUT task must make conditional rendezvous with the FURNACE tasks.

Exceptions may occur at different places, such as during the rendezvous between FURNACE and THERMOMETER. The OUTPUT task may have to accept both SEND calls and SEND_ERROR calls, which requires a select statement.

Finally, by assuming that there are more than one thermometer but not enough to go around, you may illustrate guards. (An allocator task accepts ACQUIRE only when inulase is less than THERMO_NO, for example.)

It is fairly easy to provide the "scaffolding" tasks that simulate the mainframe communication and write a log on the screen as events occur.

The example has the advantage that it makes realistic use of concurrency. I discuss it in a my paper "Entity-life modeling and structured analysis in real-time software design – a comparison" (CACM Dec 1989). The step-wise development of the example is included in my forthcoming book "Software construction in Ada". I'll be happy to provide draft copies of that chapter to anyone interested.

Dr. Bo Sanden, ISSE

George Mason University

Fairfax, VA 22030-4444

bsanden@gmuvax.gmu.edu

3.2 Modifications and Development Strategy

This example was modified slightly and the development was split into three stages.

1. a. There is one THERMOMETER which can be used at any time and returns a constant value. This is implemented using a procedure.
   b. There is a MAINFRAME which outputs a message when a reading is received from one of the FURNACES. The readings from each FURNACE are merged. The message only indicates that a reading has been received and not its contents. Readings are passed between objects using lazy streams.
   c. A fixed delay is used for controlling the sampling time.
   d. No facility is provided for changing the sampling time of the furnaces.
2. a. The THERMOMETER is protected so that only one FURNACE may use it at a time. The temperatures it returns increase uniformly.
   b. The output messages indicate the source of the reading and its value.
   c. The delay for controlling the sampling time is based on the current clock value.
3. The sampling time of the furnaces is changed at regular intervals so that the rate of sampling decreases.

Error handling and error messages were not implemented because the DAD preprocessor does not translate exceptions and exception handlers correctly. The multiple thermometers option was not implemented because this is best solved using standard Ada constructs.

3.3 Outline of the DAD program

The main components of the DAD program are as follows:

1. The THERMOMETER which is controlled by a task and whose value is accessed via a function call:

```
task THERMOMETER is
  entry TEMPERATURE (TEMP: out INTEGER);
end THERMOMETER;
function GET_TEMPERATURE return INTEGER;
```

2. The MAINFRAME's control of the FURNACEs' sampling rates, which is done via streams between the MAINFRAME and each FURNACE:

```
package INTERVAL_STREAM is new STREAM_PACK(INTEGER);
type INTERVALS is array (1..NUM_FURNACES) of
     INTERVAL_STREAM.STREAM;
function MAINFRAME_INPUT return INTERVALS;
  function INPUT_GENERATION(DUE : TIME; NEW_VALUE : INTEGER)
     return INTERVAL_STREAM.STREAM;
```

3. The generation of results by each FURNACE:

```
type READING is record
  DATE : TIME;
  TEMP : INTEGER;
end record;
package READING_STREAM is new STREAM_PACK(READING);
function SAMPLE (INT : INTEGER;
                 INPUT_STREAM : INTERVAL_STREAM.STREAM)
         return READING_STREAM.STREAM is
  function SAMPLE_DUE (INT : INTEGER; DUE : TIME;
                       INPUT_STREAM : INTERVAL_STREAM.STREAM)
         return READING_STREAM.STREAM;
```

4. The collation of the results from each FURNACE:

```
function DATE_BASED_MERGE (S1, S2 : in READING_STREAM.STREAM)
         return READING_STREAM.STREAM;
```

5. The display of the results by the MAINFRAME:

```
procedure MAINFRAME_OUTPUT (OUTPUT_STREAM:
                    in out READING_STREAM.STREAM) is
  procedure OUTPUT(R : READING);
```

The complete DAD code and the generated Ada code for the final version are shown in Appendix C. An earlier version of this code had to be modified to allow the DAD preprocessor to translate it correctly. A comment in the DAD code notes the modification required.

It was relatively easy to develop the DAD code for the case study, although difficulties were encountered due to translation errors produced by the preprocessor and deficiencies in the compilers (discussed in Sections 4 and 6.3).

4 Preprocessing, Compiling, Linking and Executing

The DAD programs and the case study were analysed to determine if there were any problems generating executable code from DAD source code. This involves several steps. Firstly, the DAD source code is translated into Ada source code using the `dad` command. This code is compiled and linked using a standard Ada compiler and linker to produce a UNIX executable. This is illustrated in Figure 1. The resultant executables are run to see if they work correctly.

Three different Ada tool sets were used in the evaluation process. The demo systems, test programs and the case study were all compiled and linked using the Telesoft (V4.1) and VAX Ada tools. In addition, some of the code was fed through the Verdix (V5.5) tools.

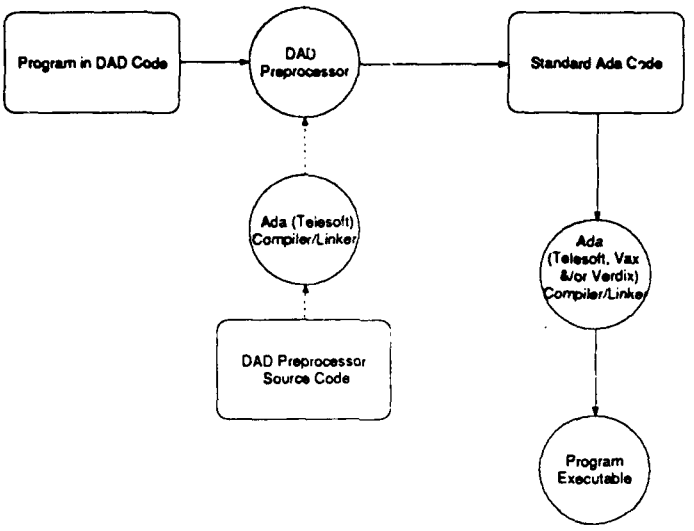

Figure 1 A Diagram of the Processes Undertaken to Create the DAD Executables.

A variety of problems were found in preprocessing, compiling, linking and executing the various DAD programs:

1. The compilation order for the `jstupl` sample program is incorrectly documented in the `READ_ME` file.
2. The preprocessor mistranslates exception handlers, nested first-class functions and some uses of a function name within the function's body. It also seems not to generate task termination code correctly in some circumstances.
3. The DAD preprocessor generates code that makes non-portable assumptions about the lengths of Ada code lines accepted by compilers.
4. The DAD runtime system includes code which makes non-portable assumptions about task priorities.
5. All three Ada compilers have problems compiling nested generics, many of which are generated by the preprocessor. In the case of Telesoft, the compiler crashes.
6. The Verdix Ada linker has problems with some examples.
7. The implementation of TEXT_IO.OPEN in the Telesoft runtime system appears not to work with tasking.
8. The Telesoft debuggers are incapable of debugging any code that involves generics.

The problems have a variety of sources: problem 1 is a documentation error and problems 5–8 are deficiencies in the Ada compilation systems. "Normal" Ada portability problems were also encountered with some of the Ada compilers supporting nonstandard `math` libraries. The remaining problems are associated with the DAD preprocessor and were all encountered while developing the case study. Problems 3 and 4 were also encountered when the delivered programs were being evaluated.

It is worth noting that some of the DAD preprocessor errors are fairly basic, and showed up during routine programming. This is indicative of the level of maturity of the DAD tools in general.

5 Translation and Performance Characteristics

This section examines certain compilation and runtime performance characteristics of DAD. Section 5.1 examines the preprocessor execution time and the size of the generated Ada files. Section 5.2 deals with DAD's runtime characteristics, concentrating on its use of stack and heap space. The execution speed of DAD code was not examined in detail in light of the current level of maturity of the DAD system.

5.1 Preprocessor Properties

This section looks at some properties of the DAD preprocessor, in particular the time it takes to translate a file and the relative size of the resultant Ada files. The graphs use the program numbers given in Table 6, with the number 16 to representing the case study. The preprocessor times and resultant file sizes are for the entire programs (excluding DAD libraries). A more objective file size measure than "lines of code" was obtained by stripping out all comments and then counting Ada / DAD tokens. The `sed` script used to measure file sizes is given in Appendix A.

5.1.1 Preprocessor Time

This section looks at the time taken to preprocess DAD files. Of particular interest is how the preprocessor time depends on the size of the DAD program.

Timing measurements were made on a single user SUN SPARC IPC with 24Mb of memory running SunOS 4.1.1 and OpenWindows. The DAD executables were on a remotely mounted NFS file system. The measurements are UNIX elapsed clock times, measured using the UNIX `date` system call, averaged over 10 runs.

Figure 2 plots preprocessing time versus DAD file size for the sample programs. It is difficult to draw any firm conclusions because there are no mid-range data points but the shape appears to lie between linear and quadratic. More detailed investigation would be required to determine the exact relationship and its root cause.

The Ada compilation times for the generated files were found to be on average 5 times as long as the preprocessing time. Thus, DAD preprocessing does not have a significant impact on overall compilation times. However, it was not possible to compare these times to the compilation times for equivalent programs written in Ada.

5.1.2 Code Expansion

This section compares the size of DAD files to the corresponding generated Ada files. This gives some indication of the succinctness of DAD compared to Ada for this style of programming. However, a comparison with equivalent hand written Ada code might give different conclusions.

Figure 3 plots the code size of DAD files against the size of the generated Ada code. This graph appears to be linear, though sample program #15 is significantly off the line of best fit. This is probably because much of the cruise control problem involves straight forward arithmetical calculation. The remaining DAD programs are approximately half the size of the generated Ada code.

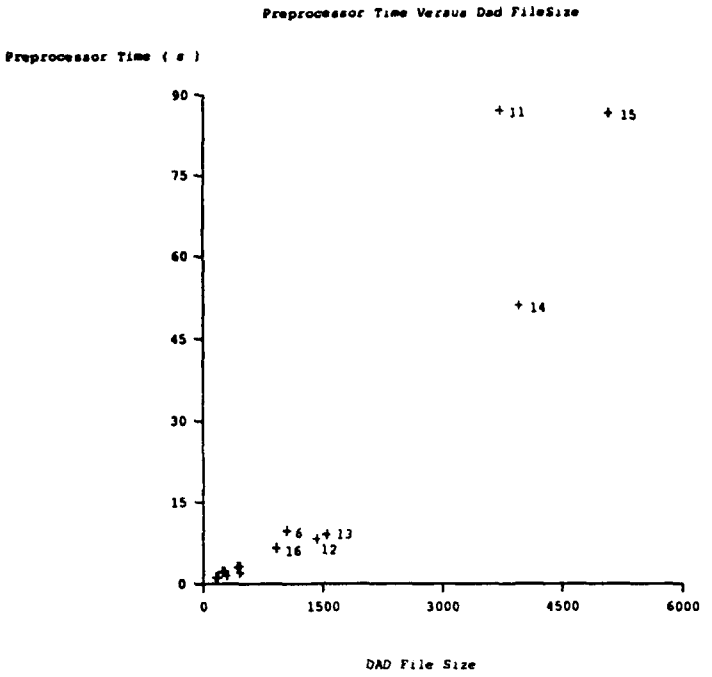

Figure 2 Graph Showing the Time Taken by the DAD Preprocessor on Files of Various Sizes.

5.2 Runtime Performance Characteristics

This section looks at the runtime memory characteristics of DAD using "micro" benchmark programs.

The programs were designed:

1. to compare the storage requirements for assigning first-class functions to variables where optimisations either could, or could not, be made;
2. to compare the optimisation of a single recursive first-class function to that for mutually recursive first-class functions and
3. to compare these optimisations to those of other functional languages.

5.2.1 Background

It is a general property of a language with first class functions is that it is not possible to use a simple stack to hold function arguments and local variables. This is because variables may still be accessible via first class function values when the enclosing scope is exited. For example:

```
procedure gen_adder () : procedure() : int
      count := 1
     procedure add () : int
        count := count + 1
        count
     add
let adder = gen_adder()
let res = adder()
```

In the above example, the space used to hold the `count` variable cannot be discarded when a call to `gen_adder` completes. Instead the space must be retained as long as the function value which has been assigned to `adder` is accessible by the program.

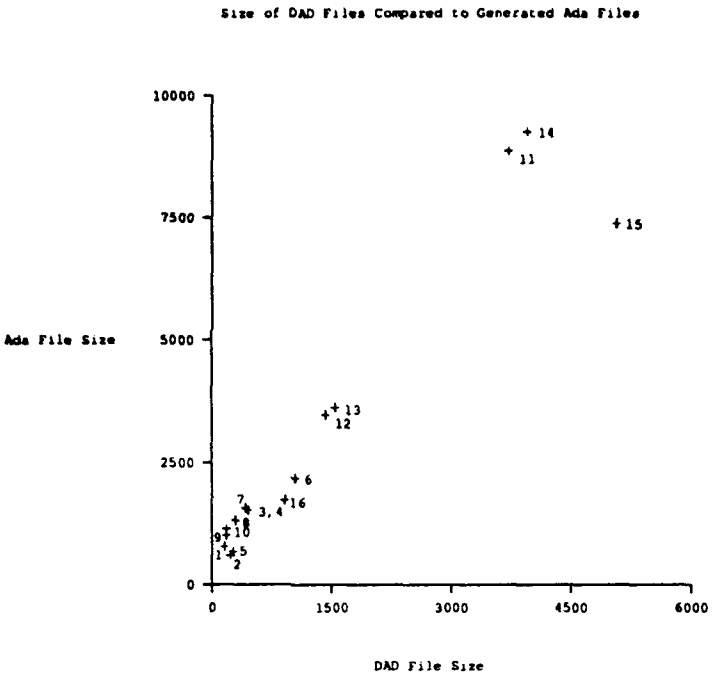

Figure 3 Graph Showing the Relationship Between the Sizes of the .d Files and Corresponding .ada Files.

The DAD system uses task instances to implement first class functions [9]. There is a task for each function type, and new tasks are generated when a new function value is created or a first class function is called. These tasks serve two purposes. Firstly, they encapsulate the scope of a function within the body of a task so that nested first class functions can access their local variables and arguments after the function has exited. Secondly, they get around limitations of the Ada type system in order to pass classes of first class functions as arguments and results. DAD's lazy values are handled in a similar way.

The use of tasks to implement function values and function calls will generally lead to substantial runtime overheads. However, there is considerable scope for optimisation to reduce these costs. The possible optimisations fall into two categories; those which are applicable to all functional languages, and those which relate specifically to DAD's use of tasking.

A common practice in functional programming is to implement algorithms for list and tree structures using recursive function calls. For example, the following function searches for a given number in a list.

```
rec type Int_list ( is_empty : bool ) is record
         case is_empty is
            when true => empty : nil
            when false => cons : record( head : int;
                                         tail : List[t] )
         end case;


rec procedure search ( l: Int_list, target: int ): bool is
         if l.is_empty then false
         else if l.cons.head  = target then true
         else search( l.cons.tail )
```

Close examination of the above function shows that the last thing done by a call to `search` in most cases is a recursive call to itself. This is referred to as "tail recursion". As a general rule, tail recursion can be optimised by replacing the call with code to set up the call's new arguments followed by a branch to the start of the function. The function's arguments and variables are effectively reused, so that for the example, the stack space overhead is constant rather than being proportional to the length of the list.

A second general optimisation involves analysing functions to see if their variables and arguments need to be allocated in dynamic space. For example, if a scope does not contain any code that generates function values, its variables can be allocated on the stack.

The DAD specific optimisations are mainly in the area of resource reclamation. These are necessary largely because typical Ada systems are not garbage collected and do not reclaim orphaned tasks. The DAD preprocessor therefore needs to generate special code for reclaiming tasks where possible. DAD primitive constructs that are implemented in Ada can also be hand optimised in various ways.

5.2.2 Methodology

The methodology used to examine the runtime behaviour of DAD programs was to write some small benchmark programs, execute them to measure their space utilisation, and examine the generated Ada code to try to understand the differences in the results. The complete set of benchmark programs along with the corresponding generated Ada code is given in Appendix B.

A fairly primitive technique was used to measure space utilisation. Sun UNIX provides the user with the ability to set upper limits on a program's use of certain resources; in particular the data size and the stack size. The shell built-in `limit` function[1] was used to set these and the benchmarks were run to see how long they ran. Each benchmark outputs one or more lines of text for each iteration, so progress can be measured by counting the lines. The stack and data resource limits under the configuration used, have a granularity of 4096 bytes and 8192 bytes respectively, but this has negligible effect on the actual measurements.

It is worth noting that the benchmarks did not always terminate in the correct way under some Ada implementations tested. The two programs which were dependent on the data limit terminated with a "bus error" or "segmentation violation" rather than with an Ada STORAGE_ERROR exception as required by the Ada Reference Manual.

5.2.3 First-class Function Generators and Variables

The `function_generator_and_call` program is designed to examine the storage overhead of first class function instances. It creates a large array of functions and loops to initialise the array elements with a trivial function value:

```
type FN is function  return INTEGER;

type FN_GEN is function (I:INTEGER) return FN;

type FN_ARRAY is array (0..100000) of FN;
FNS : FN_ARRAY

I : INTEGER;
```

[1] The Sun manual page for limit(1) states that the `datasize` limit covers both the stack and data segments. In fact, experiments indicate that it only covers the data segment, as is implied by its name.

```
  -- GENERATOR generates functions of type FN
  function GENERATOR : FN_GEN
    function : FN is
    begin
      return I;
    end;

begin
  -- create functions & output there value
  for I in FNS'RANGE loop
    FNS(I) := GENERATOR(I);
    put_line(INTEGER'IMAGE(FNS(I)()));
  end loop;
  ...
end;
```

The variable_recursion program is designed to examine the storage overhead of calls to first class functions using function variables. It is a nonterminating recursive loop where the recursive function call is made via a variable F:

```
  type FN is function  return INTEGER;

  F : FN;

  function FOO : FN;

  function FOO : FN is
    begin
      F := FOO;
      put_line("In Foo");
      return F();
    end;
  function DUMMY: FN is
    begin
      return 1;
    end;

begin
  F := DUMMY;
  FOO();
end;
```

The function_generator_and_call program generates function values and then calls them. The function_generator program is a variant of function_generator_and_call, which avoids the overhead of the final function calls, replacing the original putline statement with:

```
    put_line("In Loop");
```

Figure 4 shows the UNIX data segment usage for each program against the number of function values or instances created. The number of loop or recursive iterations before the programs halted was found to be independent of the stack segment limit. This simply indicates that the recursion in variable_recursion is controlled by a task or tasks rather than the main program.

The graphs show that the data space used by function_generator_and_call, variable_recursion and function_generator is a linear function of the number of function values or instances created. The slopes of the graphs show that the generation of each function value requires 31.8Kb and 31.2Kb in func-

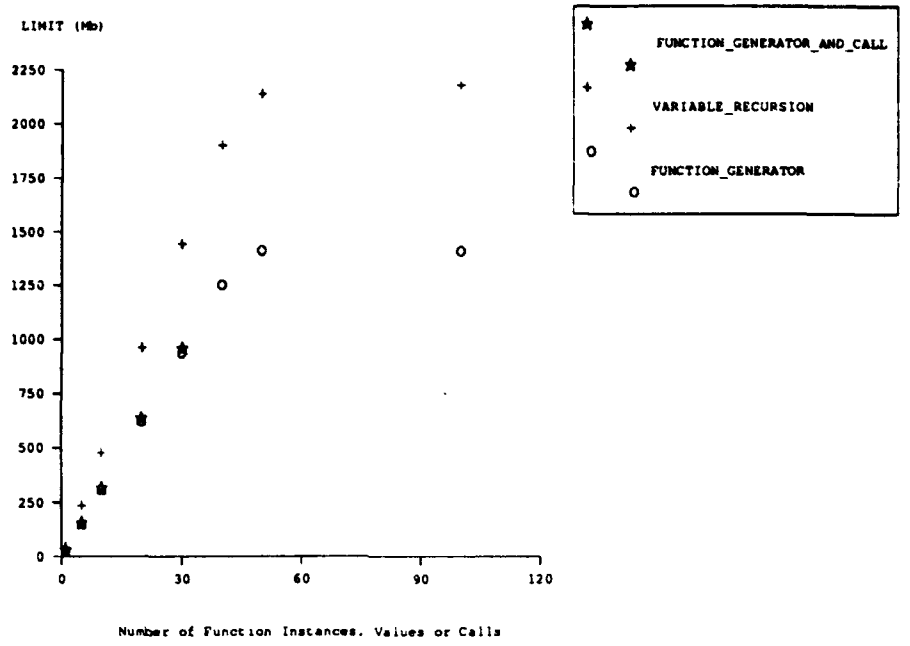

Figure 4 Number of Function Values or Instances versus the Data Segment Usage

tion_generator_and_call and function_generator respectively. The extra 0.6 Kb used in function_generator_and_call could be due to the additional function call or it could be an artifact of the granularity of the measurement technique. (Note that it is difficult to determine the overhead of function calls because simple calls are optimised.)

The slope of the graph for variable_recursion shows that each new function instance requires 20.7 Kb. This is noticeably less than the amount for memory required for each function value in function_generator_and_call and function_generator, despite the relative simplicity of the functions.

The "knee" in the graphs beyond the ~50Mb point indicates that another system parameter is affecting the results for large data segment limits: in this case, the parameter is most likely the amount of physical swap space available on the system.

5.2.4 Recursive First-Class Functions

The simple_recursion program is designed to examine the storage overhead of recursive first class function calls. The FOO function is a simple nonterminating tail recursive loop:

```
type FN is function  return INTEGER;

function FOO : FN is
-- recursive first-class function
begin
    put_line("In Foo");
    FOO();
end;

begin
```

```
  FOO();
end;
```

The mutual_recursion program is intended to examine the storage overhead of mutually recursive first class function calls. The FOO and DUMMY functions are used.

```
  type FN is function  return INTEGER;

  function DUMMY : FN;

  function FOO : FN is
  -- mutually recursive with DUMMY
    begin
      put_line("In Foo");
      return DUMMY();  -- mutually recursive call
  end;
  function DUMMY: FN is
  -- mutually recursive with FOO
    begin
      put_line("In Dummy");
      return FOO();  -- mutually recursive call
    end;

begin
  FOO();
end;
```

Figure 5 shows the UNIX stack usage for each program against the number of recursive function calls in progress. The graph indicates that the relationship is linear and that each recursive function call requires only 141 bytes of stack space for both programs.

The data segment limit had no effect of the number of function calls. Examination of the generated Ada code indicates that the DAD preprocessor has deduced that the functions create no function values and have no arguments and therefore has optimised away the tasks normally used for performing first class function calls. No tail recursion optimisation has been performed at this level. The stack usage graphs suggest that the same is true for the Ada level.

5.2.5 Conclusions

The DAD preprocessor performs some optimisation: when a function is not required to be first-class or takes no arguments, the preprocessor does not generate any tasks for instances of the function. A number of other desirable optimisations are not included in the DAD compiler. The DAD system can make use of optimisations performed by Ada compilers (when the appropriate options are selected); for example, the compiler may be able perform some tail recursion optimisation. However, many of the possible optimisations are the responsibility of the preprocessor; for example, an Ada compiler cannot be expected to optimize tail recursion in first-class functions implemented using tasks.

The storage overheads for unoptimised first-class function instances are approximately 200 times the overhead for optimised function calls. The functions FUNCTION_GENERATOR_AND_CALL, VARIABLE_RECURSION and FUNCTION_GENERATOR all have storage overheads greater that 20 Kb per function compared to 141 bytes for SIMPLE_RECURSION and MUTUAL_RECURSION where the optimisation is performed. The benefits of optimizing whenever possible appear to be substantial, but only a limited number of optimisations are performed.

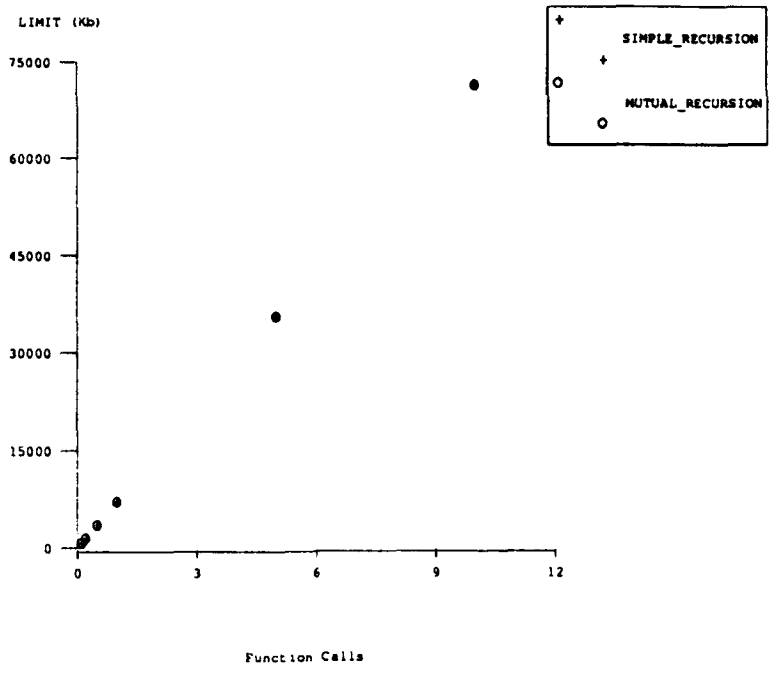

Figure 5 Number of Function Calls versus Stack Usage

6 Benefits and Limitations

Many of the delivered papers claim that DAD is a useful language for rapid prototyping [8, 4, 9]. Others, in particular "An Evaluation of the Declarative Ada Dialect", claim that it is not a general purpose rapid prototyping language, but is useful for Ada programmers[11, 12]. This section considers the benefits and drawbacks of using DAD for rapid prototyping, focusing mainly on development where the final implementation of the system must be in Ada. In particular we look at

1. DAD compared to other functional languages,
2. the compatibility of DAD with standard Ada,
3. the robustness and maturity of the DAD language and preprocessor and
4. the portability of the DAD implementation strategy and physical implementation.

6.1 DAD and Other Functional Languages

There are two main points of comparison between DAD and other functional languages which are used for rapid prototyping. These are the languages' succinctness and the performance of their implementations.

Succinctness

DAD was designed to be Ada-like, so its syntax and constructs are less concise than those of other functional languages. This is highlighted by the following quotes from Document 11 (written by a semi-independent reviewer of DAD):

> "The drawback to DAD's busy syntax is that it detracts heavily from its usefulness as a rapid prototyping language. More effort must be invested merely in expressing ideas in the language, and alterations and rewrites become increasingly tedious."

As well as having a more verbose syntax than other functional languages, a number of features present in most functional programming languages are absent from DAD.

> "The absence of pattern matching and "natural" currying also does much to increase the verbosity of DAD . . . Furthermore, the need to explicitly write a function in curried form tends to obscure the possibility of using currying . . . the idea to use a partially parameterized function may well occur after it is written."

DAD provides a substitute for "natural currying" which enables programs to explicitly declare functions which can be partially parameterized.

> "It is in polymorphism, however, that Miranda has its great advantage."

It should be noted that DAD has limited polymorphism available through Ada generics. However that offered by other functional languages is more powerful and less verbose.

Performance

The DAD preprocessor generates Ada code that relies heavily on Ada tasks and generics. Both of these constructs have high time and space overheads. In addition, the DAD preprocessor only performs a subset of the optimisations performed by other functional languages, resulting in further time penalties.

Functional programmers often write programs for succinctness. The compiler is then responsible for ensuring the source code gets translated into an efficient implementation. Poor performance often results in systems where the compiler does not make appropriate optimisations. As the DAD compiler performs only a limited subset of the common optimisations, the use of DAD can incur time penalties which are not possessed by other functional languages, and which may be unacceptable in many applications.

DAD's usefulness as a rapid prototyping tool is limited when compatibility with Ada is not required. If other functional languages are available, their use would be preferable.

6.2 DAD and Ada

As DAD is an extension of Ada it may offer some benefits in an Ada environment. There are several things to consider:

1. How well do the syntactic and semantic extensions provided by DAD fit into the Ada language?
2. Can Ada components be used in DAD programs?
3. Can DAD components be used in Ada programs?
4. How easy is it for Ada programmers to understand DAD programs? and
5. How easy is it for Ada programmers to learn to write DAD programs?

DAD Extensions of Ada

The syntactic extensions provided by DAD are well documented in the delivered papers, in particular [6]. In contrast, the documentation of DAD's semantics is limited; the sematics must mainly be inferred from the examples or the implementation. There are a number of issues regarding the use of DAD which could not be considered due to a lack of semantic information (the current implementation is too immature to use as a basis for analysis), for example:

1. the interaction of first-class functions with Ada tasks,
2. the use of first-class functions in generic instantiations,
3. the effect of garbage collection (or lack of it) on program development.

In addition, the advent of Ada9X will raise other issues of compatibility.

These and other related issues are complex to address and remain unresolved.

Ada Components in DAD Programs

DAD programs can make use of standard Ada libraries and other reusable Ada components: for example, the standard Ada libraries `Calendar` and `Text_IO` were used in the DAD code developed as part of the case study. However, Ada packages which use *limited private* types require adaptations to enable them to be used in DAD programs. This is because functions may

only have *in* arguments, but objects of limited private types must be passed as *in out* parameter. Simple buffer packages may be used to overcome these problems. The packages contain new *access* types which reference the original limited private types.

DAD Components in Ada Programs

DAD libraries and components can also be used by Ada programs, provided the Ada programs are preprocessed to translate the calls to DAD functions and the references to the DAD lazy types and lazy objects.

The DAD library `stream_pack` as well as other lazy types may be useful additions to a repository of reusable components. Indeed the `stream_pack` library proved very useful during the development of the case study. It is a hand-coded package which implements generic lazy streams, where instances of streams may have multiple producers and consumers. When the full generality of these streams are needed the package should provide efficient code. However, for lazy streams with only one producer or consumer, other, more efficient implementations may be possible.

Other lazy types can be created using the DAD language constructs and the DAD preprocessor can then be used to generate Ada code. As the implementation of lazy types requires some form of concurrency control, DAD versions may generate satisfactory Ada implementations.

Using DAD to construct lazy types should decrease their development time, since DAD code is simpler to write than code which directly manipulates tasks and is also shorter. (The expansion ratio for DAD to generated Ada code is approximately 4:1.) In general this brevity is at the expense of performance, but hand-coding can produce more efficient code — as illustrated by the library `stream_pack`.

Understanding DAD Programs

DAD programs use an Ada-style notation. Thus, programmers familiar with Ada should, in theory, be able to read and understand simple DAD programs. Some programs which use lazy types and first-class functions (including programs which are straight-forward applications of techniques well-understood by functional programmers) may initially cause some difficulties.

As DAD programs are more succinct than their generated Ada counterparts, some DAD programs may be understood more easily than equivalent Ada programs.

Writing DAD Programs

Ada is a procedural language. DAD was developed as an Ada-style language which supports the functional programming paradigm. The functional programming paradigm is significantly different from the procedural programming paradigm.

Ada programmers with no experience in functional programming cannot be expected to write well-structured functional-style programs. The use of DAD for program development does not guarantee the quality of the resulting programs. In fact, as DAD is a super-set of Ada, Ada programmers are more likely to stick to the standard Ada subset with only limited use of the DAD constructs.

The difference between the functional and procedural programming paradigms is seen as a major impediment to the uptake of DAD in Ada environments. The difficulty in learning to program in a functional language lies not with the syntax of the language, but rather with the underlying concepts and techniques. The argument the DAD developers used to support the development of DAD is debatable. They say that:

1. "the more new technology is presented as an increment to an existing one, the less learning is necessary" and
2. "natural human resistance to innovation needs to be overcome by avoiding unnecessary change" [4]

We believe that these are some of the main reasons why the DAD features will *not* be adopted — even if its use was mandated within an organisation. The presence of all the Ada concepts in the

new language enables Ada programmers to program in standard Ada. It is precisely those who are resistant to change who will continue to program in Ada.

6.3 Robustness and Maturity

The previous discussions in this section assume that the DAD language and preprocessor are robust. However, as mentioned in Section 4, a number of problems, including incorrect translations, were encountered while using the DAD preprocessor during the implementation of the case study.

Several of the incorrect translations lead to runtime errors which were difficult to trace with some of the Ada systems. The debugging of generic instantiations was not supported on one system, while another propagated tasking exceptions when other exceptions arose earlier and should have been propagated.

The incorrect translation of DAD exception handlers was particularly annoying, since exception handlers would have been useful in determining some of the other incorrect translations.

Errors occurring during the compilation or preprocessor phases were relatively easy to relate to the DAD source code, despite the simplistic error tracking procedures available. The only supported method for tracing errors detected during the Ada compilation to DAD coding errors is a mapping between the DAD and Ada lines. (See Document 3 in [2].)

In summary, the DAD preprocessor cannot be considered as mature or robust. It still produces translation errors for simple constructs and it was easy to produce new programs which were translated incorrectly. The maturity of the language is harder to assess. However the discussions above indicate areas where it could be improved.

6.4 Portability

The portability of DAD is limited by

1. The use of Ada compiler dependent features in the generated code and the DAD runtime system.
2. The preprocessor's inherent dependence on the file system used.

While the DAD preprocessor must be file system dependent, its implementation has not been developed with due consideration for portability. Its use of hidden directories and lack of encapsulation of the I/O make porting more arduous than it need be.

More important is the DAD preprocessor's use of Ada tasks, priority and generics. The implementation and scheduling of tasks are compiler dependent. This can result in different compilation problems and run-time behaviour for different compilation systems.

DAD's heavy use of tasking and generics can also cause Ada compiler dependent limits to be violated. For example, breaches due to the excessive nesting of generics were common during this evaluation. While the DAD developers are not responsible for the current state of Ada compilers, they should acknowledge the compilers' known limitations. The DAD preprocessor should have been designed to function correctly within them.

7 Conclusions

The DAD language and its implementation can be viewed both as:

1. an experiment to demonstrate to the Ada community that Ada should be extended to support the functional programming paradigm, and
2. an experiment in defining an Ada compatible prototyping language.

7.1 DAD as a Demonstration to the Ada Community

The DAD language and preprocessor have shown that the Ada language can be extended to support the functional programming paradigm by adding first class functions and lazy types. The preprocessor demonstrates that first class functions can be successfully implemented using the existing Ada tasking mechanisms.

In spite of the above points, the DAD work is not likely to succeed in the broader aim of convincing the Ada community that Ada should support the functional programming paradigm:

1. The DAD work does not demonstrate that adding support for the functional paradigm improves the Ada language. Some of the main benefits of pure functional programming [13] are: 1) programs are easier to understand and reason about as they do not contain assignment statements, 2) it encourages thinking at higher levels of abstraction, 3) it aids automatic parallelisation due to the absence of side-effects, 4) it is the basis for many applications in Artificial Intelligence, 5) it is valuable in developing executable specifications and prototype implementations, and 6) it provides a simple framework for studying computer science. It is clear that many of these benefits are reduced in languages such as DAD which allow assignment.

   In addition, DAD supports, and in some cases requires, a mixture of functional and procedural programming. Little research has been done in this style of programming, and is not clear whether (well-written) DAD programs using a mixed paradigm programs would be easier to understand than equivalent pure Ada programs. In particular, there is reason to believe that mixed paradigm programs that mix lazy types and conventional Ada tasks would be very difficult to understand.

2. Specifying the semantics of DAD using prototypical translations of DAD constructs into Ada does not give the reader much confidence that interactions between the semantics of DAD constructs are well understood. In their defence, the DAD researchers were hampered by the lack of a formal specification for Ada.

3. In our opinion, the approach that the researchers took to implementing DAD is not likely to convince the Ada community that the functional programming paradigm can be supported without a substantial performance penalty. The main problems are the assumptions that DAD makes about the implementations of dynamic storage management and tasking in the underlying Ada runtime systems:

   a. The DAD implementation strategy makes extensive use of tasking, with closures implemented using dynamic task creation and first class function calls implemented using task switching. In practice, dynamic task creation and task switching are more expensive in runtime resources compared with the techniques normally used to implement first class functions. Even with a specially tuned implementation of Ada tasking, we would expect the cost of a DAD style first class function call to be at least an order of magnitude slower than either a conventional Ada function call or a first class function call implemented in the conventional way.

   b. The DAD implementation strategy seems to assume that tasks are garbage collected by the Ada system. Without a garbage collector, trivial DAD programs can consume vast amounts of memory very quickly. The problem is that very few Ada implementations include garbage collectors, probably because there is little demand for them. In our experience, most Ada programmers view garbage collection as very costly in machine resources, and require hard evidence to convince them otherwise.

   We acknowledge that the DAD researchers never aimed to produce a DAD implementation that is competitive with mature implementations of Ada or functional languages. Furthermore, they clearly did not have the resources needed to do so. However, given the emphasis that the Ada community places on efficiency, we believe that a different approach would have been more appropriate.

7.2 DAD as an Ada Compatible Prototyping Language

The second view of DAD is of an Ada compatible prototyping language. Once again, the DAD has succeeded in showing that some linguistic features that are appropriate to prototyping can be added to Ada. The example DAD programs supplied by UQ and those developed by the authors show that DAD can be used for prototyping small-scale problems that do not require much functional style computation.

When DAD is compared against other prototyping languages such as Smalltalk, ML and Miranda, it can be seen to have drawbacks in the following areas:

1. Succinctness: A general requirement of prototyping languages is that they allow algorithms to be expressed clearly and succinctly. Since the syntax of DAD is a superset of standard Ada, DAD programs are generally more verbose than equivalent programs in other prototyping languages. This can be attributed to Ada's 1) being statement oriented rather than expression oriented, 2) requiring explicit types in all declarations, and 3) having a generally wordy syntax.
2. Currying: Most functional languages support partial instantiation of the arguments of a higher order function by currying. In DAD, the programmer has to resort to explicitly defining a new function to do this.
3. Polymorphism: Most prototyping languages support polymorphism using both static and dynamic typing. In many such languages, polymorphism is further enhanced by subtyping or inheritence. In DAD, as in Ada, the only form of polymorphism available is through generics.
4. Storage Management: Nearly all prototyping languages try to hide the problems of dynamic storage allocation and deallocation from the programmer by providing automatic reclamation of garbage. In most cases, the programmer can ignore the issue entirely when building a prototype.

   In DAD, the problem of storage management largely falls on the programmer in practice. While it is possible to ignore the issue in "toy" problems, this approach is not viable for medium to large scale prototypes. Hand coding storage management is labour intensive and error prone in general, and will be especially so for programs written using the functional paradigm.

   The DAD authors have argued that this is a problem with the current DAD implementation that will go away in a garbage collected implementation. We believe it would have been better to have written the DAD language definition to say that a full implementation *requires* garbage collection.

One of the main potential advantages of DAD compared to other prototyping languages is its ability to make use of pre-existing Ada code. It is hard to say what impact a complete, stable implementation of DAD would have in this area. The DAD authors have recognised that it is necessary to write "wrappers" for Ada private types to allow them to be used in a functional way. It is not clear whether other changes would need to be made for more complex software to allow it to be used with DAD.

The DAD authors suggest that since DAD is compatible with Ada, it would be possible to obtain production quality Ada software by refining a DAD prototype. While this may be true, it has not been demonstrated that this approach is practical. We think that DAD code that makes extensive use of lazy types is likely to be especially difficult to transform.

The testing and analysis of the DAD system carried out in SE group have clearly demonstrated that the current implementation is not sufficiently mature for it to be used for serious prototyping work. The main problems identified are as follows:

1. DAD bugs: In a couple of cases, the DAD preprocessor has shown to generate incorrect Ada code. In particular, both nested first class function definitions and exception handlers are translated into Ada code that does not compile correctly. In addition, the DAD preprocessor and runtime system make non-portable assumptions about target Ada compilers and the implementation of tasking on target Ada systems.
2. Target Ada system bugs: The DAD preprocessor generates code that relies heavily on nested generics and Ada tasking in ways that are radically different from a conventional Ada program.

This has shown up a variety of problems in the 3 Ada systems used as targets. These ranged from compiler imposed restrictions on the number of levels of nesting of generics (VAX Ada), through to compiler and runtime system crashes, and totally inadequate support for debugging generics (Telesoft Ada).

While the DAD authors should not be blamed for these problems, this is a weak link in their implementation strategy. Notwithstanding the efforts of AJPO in validating Ada compilers, it is well known that many vendors' offerings are distinctly second rate, especially in infrequently used aspects of the Ada language.

3. Performance: The test results reported in this paper clearly show that the current implementation of first class functions in DAD is very "memory hungry". Inspection of the generated Ada code shows that simple optimisations normally performed by compilers for functional languages have not been performed. This, and other evidence, suggests that use of the current DAD implementation for serious prototyping work is likely to run into major performance problems.
4. Ease of use: The authors' experience in using the current DAD implementation to develop the furnace controller example is that it is not easy to use. The main problems are:
   a. Bugs and shortcomings in the DAD and Ada systems mentioned above.
   b. Difficulty tracking Ada compilation and execution errors back to the DAD source code.
   c. The lack of suitable debugging tools in general and under Telesoft in particular.
5. Incomplete testing: Judging from the nature of some of the problems found during testing and analysis, the authors suspect that the DAD preprocessor has not been subject to methodical testing, or large scale ad-hoc testing. This concern should be addressed before DAD is used for serious prototyping work.

In summary, the authors do not believe that the current implementation of DAD is suitable for serious medium or large-scale prototyping. Even assuming that the concerns relating to the current implementation are addressed, it is not clear that DAD would be superior to existing prototyping languages in the Defence context

The advantage of being able to interface DAD directly with Ada code should be measured against DAD's shortcomings as a prototyping language. It may be more fruitful to pursue alternative approaches such as providing support for interlanguage calls between Ada and Smalltalk, ML or Miranda.

The DAD software delivered by the University of Queensland is held by SE group. Any queries concerning the DAD system should be directed to the authors or the Head of DSTO's Software Engineering Group.

Acknowledgments

We would like to thank Stefan Landherr for motivating this study and his assistance in the preparation of this paper.

REFERENCES

[1] ANSI. *Reference Manual for the Ada Programming Language*, 1983.

[2] Paul A. Bailes, Dan Johnston, Eric Salzman and Li Wang. DAD - an Ada Preprocessor for Functional Programming.

[3] Gina Kingston. A Reading Guide for the Declarative Ada Dialect (DAD). ITD Divisional Paper ITD-92-19, Oct 1992.

[4] Paul A. Bailes, Dan Johnston and Eric Salzman. Rationale for the Design of an Ada Prototyping Language. Technical report, No. 228. Language Design Laboratory, Key Centre for Software Technology, Department of Computer Science, University of Queensland, August 1992.

[5] Paul A. Bailes, Dan Johnston and Eric Salzman. DAD Defined. Technical report, No. 229. Language Design Laboratory, Key Centre for Software Technology, Department of Computer Science, University of Queensland, August 1992.

[6] Paul A. Bailes, Dan Johnston, Eric Salzman and Li Wang. DAD Canonical Specification. Technical report, No. 236. Language Design Laboratory, Key Centre for Software Technology, Department of Computer Science, University of Queensland, August 1992.

[7] DAD Actual Translation Specifications (Document 5). *DAD - an Ada Preprocessor for Functional Programming*, August 1992.

[8] Paul A. Bailes, Dan Johnston, Eric Salzman and Li Wang. Full Functional Programming in a Declarative Ada Dialect (Document 1a). *DAD - an Ada Preprocessor for Functional Programming.*

[9] Paul A. Bailes, Dan Johnston and Eric Salzman. First-Class Functions for Ada. Technical report, No. 225. Language Design Laboratory, Key Centre for Software Technology, Department of Computer Science, University of Queensland, August 1992.

[10] DAD Demonstrations (Document 6). *DAD - an Ada Preprocessor for Functional Programming.*

[11] Paul A. Bailes, Dan Johnston and Eric Salzman. Prospects for Ada Language and Technology R&D (document 12). *DAD - an Ada Preprocessor for Functional Programming.*

[12] Mark Pedersen. An Evaluation of the Declarative Ada Dialect (Document 11). *DAD - an Ada Preprocessor for Functional Programming*, August 1992.

[13] Bruce J.MacLennan. *Functional Programming - Practice and Theory.* Addison-Wesley, 1990.

APPENDIX A
SED Script for Determining Ada and DAD tokens

The following SED Script was produced to count the number of tokens in an Ada or DAD source file. Comments are not counted.

```
#!/bin/sh
#
# PROGRAM : SYM_COUNT
# AUTHOR  : STEPHEN CRAWLEY
# DATE    : JANUARY 29
# PURPOSE : This script can be used to count the number of tokens
#           in a DAD or an Ada source code file.
#
( sed -e 's/--.*//' \
      -e 's/^\(.*\)"\([^"]*\)""\([^"]*\)"/\1"\2!\3"/g' \
      -e 's/"[^"]*"/!/g' \
      -e 's/'\''.'\''/!/g' \
      -e 's/<>/!/g' \
      -e 's/:=/!/g' \
      -e 's/<=/!/g' \
      -e 's/>=/!/g' \
      -e 's-/=-!-g' \
      -e 's/\*\*/!/g' \
      -e 's/<</!/g' \
      -e 's/>>/!/g' \
      -e 's/\.\./!/g' \
      -e 's/=>/!/g' \
      -e 's-\([^ ]\)\([!-/]\)-\1 \2-gp' \
      -e 's-\([^ ]\)\([:-@]\)-\1 \2-gp' \
      -e 's-\([^ ]\)\([[-^`]\)-\1 \2-gp' \
      -e 's-\([^ ]\)\([[-~]\)-\1 \2-gp' \
      -e 's-\([!-/]\)\([^ ]\)-\1 \2-gp' \
      -e 's-\([:-@]\)\([^ ]\)-\1 \2-gp' \
      -e 's-\([[-^`]\)\([^ ]\)-\1 \2-gp' \
      -e 's-\([[-~]\)\([^ ]\)-\1 \2-gp' \
   $1 ) | wc
```

APPENDIX B
Source Code for Benchmarks

The sections in this appendix are the DAD and corresponding Ada files for a series of programs designed to look at the storage requirements for DAD programs.

B.1 DAD Function_Generator_And_Call Code

```
--------------------
--
-- PROGRAM :-  FUNCTION_GENERATOR_AND_CALL
--
-- AUTHOR  :-  Gina Kingston
--
-- DATE    :-  7 Oct 92
--
-- PURPOSE :-  To test the storage implications of assigning unique
--             first-class functions to elements of an array.
--
-- ** This is a Declarative Ada Dialect (DAD) file **
--
--------------------
with TEXT_IO;
use TEXT_IO;

procedure FUNCTION_GENERATOR_AND_CALL is

  type FN is function  return INTEGER;
  -- main first-class function type

  type FN_GEN is function (I:INTEGER) return FN;
  -- function type used to generate functions of first type

  type FN_ARRAY is array (0..100000) of FN;
  FNS : FN_ARRAY;
  -- the array for the functions

  I : INTEGER;


  -- GENERATOR generates functions of type FN
  function GENERATOR : FN_GEN
    function : FN is
    begin
      return I;
  end;
```

```
begin
  -- create functions & output there value
  for I in FNS'RANGE loop
    FNS(I) := GENERATOR(I);
    put_line(INTEGER'IMAGE(FNS(I)()));
  end loop;
  -- re-output the values of the functions
  -- this was included to ensure that garbage collection
  -- would not effect the storage requirements for this program
  for I in FNS'RANGE loop
    put_line(INTEGER'IMAGE(FNS(I)()));
  end loop;
end;
```

B.2 Ada Function_Generator_And_Call Code

```
with LAda_lazy , DAda_box_package ;
with DAda_functions ;
with TEXT_IO ;
use TEXT_IO ;
procedure FUNCTION_GENERATOR_AND_CALL is
    type DAda_FN_params is record
        null;
    end record ;
    DAda_FN_dummy_params : DAda_FN_params ;
    package DAda_FN_package is new DAda_functions
            ( DAda_FN_params , INTEGER ) ;
    type FN is new DAda_FN_package.funct_type ;
    procedure DAda_FN_complete renames DAda_FN_package.kill_functs ;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER ;
    type DAda_FN_GEN_params is record
        I : INTEGER ;
    end record ;
    package DAda_FN_GEN_package is new DAda_functions
            ( DAda_FN_GEN_params , FN ) ;
    type FN_GEN is new DAda_FN_GEN_package.funct_type ;
    procedure DAda_FN_GEN_complete renames
              DAda_FN_GEN_package.kill_functs ;
    function DAda_apply_FN_GEN (
        I : in INTEGER ;
        DAda_fn : in FN_GEN ) return FN ;
    type FN_ARRAY is array ( 0 .. 100000 ) of FN ;
    FNS : FN_ARRAY ;
    I : INTEGER ;
    task type DAda_GENERATOR_task is
        entry DAda_init (DAda_params: in DAda_FN_GEN_params);
        entry DAda_result (result: out FN);
    end DAda_GENERATOR_task;
    type DAda_GENERATOR_task_ptr is access DAda_GENERATOR_task;
    function GENERATOR (params: DAda_FN_GEN_params) return FN;
    package DAda_GENERATOR_replacement is new
            DAda_FN_GEN_package.funct_generator (GENERATOR);
    task body DAda_GENERATOR_task is
        I : INTEGER ;
```

```
        DAda_params : DAda_FN_GEN_params ;
        DAda_res : FN ;
    begin
        accept DAda_init (
            DAda_params : in DAda_FN_GEN_params ) do
            DAda_GENERATOR_task.DAda_params := DAda_params ;
            DAda_GENERATOR_task.I := DAda_params.I ;
        end DAda_init ;
        declare
            function DAD_func1 (
                DAda_params : in DAda_FN_params ) return INTEGER is
            begin
                return I ;
            end DAD_func1;
            package DAda_DAD_func1_replacement is new
                    DAda_FN_package.funct_generator ( DAD_func1 ) ;
        begin
            DAda_res := FN(DAda_DAD_func1_replacement.make) ;
            accept DAda_result (
                result : out FN ) do
                result := DAda_res ;
            end DAda_result ;
        end ;
    end DAda_GENERATOR_task;
    function GENERATOR (params: DAda_FN_GEN_params) return FN is
        DAda: DAda_GENERATOR_task_ptr := new DAda_GENERATOR_task;
        DAda_rslt: FN;
    begin
        DAda.DAda_init (params);
        DAda.DAda_result (DAda_rslt);
        return DAda_rslt;
    end GENERATOR;
    function DAda_apply_FN_GEN (
        I : in INTEGER ;
        DAda_fn : in FN_GEN ) return FN is
    begin
        return DAda_FN_GEN_package.apply ( DAda_FN_GEN_params ' ( I =>
                I ) , DAda_FN_GEN_package.funct_type (DAda_fn) ) ;
    end DAda_apply_FN_GEN;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER is
    begin
        return DAda_FN_package.apply ( DAda_FN_dummy_params ,
                DAda_FN_package.funct_type (DAda_fn) ) ;
    end DAda_apply_FN;
begin
    for I in FNS ' range loop
        FNS ( I ) := GENERATOR ( DAda_FN_GEN_params ' ( I => I ) ) ;
        put_line ( INTEGER ' IMAGE ( DAda_FN_package.apply
            ( DAda_FN_dummy_params ,
              DAda_FN_package.funct_type ( FNS ( I ) ) ) ) ) ;
    end loop ;
    for I in FNS ' range loop
        put_line ( INTEGER ' IMAGE ( DAda_FN_package.apply
            ( DAda_FN_dummy_params ,
              DAda_FN_package.funct_type ( FNS ( I ) ) ) ) ) ;
```

```
    end loop ;
    DAda_GENERATOR_replacement.kill ;
    DAda_FN_GEN_package.kill_functs ;
    DAda_FN_package.kill_functs ;
end FUNCTION_GENERATOR_AND_CALL;
```

B.3 DAD Simple_Recursion Code

```
--------------------
--
-- PROGRAM :-  VARIABLE_RECURSION
--
-- AUTHOR  :-  Gina Kingston
--
-- DATE    :-  4 Nov 92
--
-- PURPOSE :-  To test the storage implications of recursive
--             first-class function calls
--
-- ** This is a Declarative Ada Dialect (DAD) file **
--
--------------------
with TEXT_IO;
use TEXT_IO;

procedure SIMPLE_RECURSION is

  type FN is function  return INTEGER;
  -- type for the first-class function


  function FOO : FN is
  -- recursive first-class function
  begin
      put_line('In Foo');
      FOO();
  end;

begin
  FOO();
end;
```

B.4 Ada Simple_Recursion Code

```
with LAda_lazy , DAda_box_package ;
with DAda_functions ;
with TEXT_IO ;
use TEXT_IO ;
procedure SIMPLE_RECURSION is
    type DAda_FN_params is record
        null;
    end record ;
    DAda_FN_dummy_params : DAda_FN_params ;
    package DAda_FN_package is new
```

```
            DAda_functions ( DAda_FN_params , INTEGER ) ;
    type FN is new DAda_FN_package.funct_type ;
    procedure DAda_FN_complete renames DAda_FN_package.kill_functs ;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER ;
    function FOO (
        DAda_params : in DAda_FN_params ) return INTEGER is
    begin
        put_line ( "In Foo" ) ;
        return FOO ( DAda_FN_dummy_params ) ;
    end FOO;
    package DAda_FOO_replacement is new
            DAda_FN_package.funct_generator ( FOO ) ;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER is
    begin
        return DAda_FN_package.apply ( DAda_FN_dummy_params ,
                DAda_FN_package.funct_type (DAda_fn) ) ;
    end DAda_apply_FN;
begin
    declare
        DAD_f : INTEGER := FOO ( DAda_FN_dummy_params ) ;
    begin
        null;
    end ;
    DAda_FOO_replacement.kill ;
    DAda_FN_package.kill_functs ;
end SIMPLE_RECURSION;
```

B.5 DAD Variable_Recursion Code

```
--------------------
--
-- PROGRAM :-  FUNCTION_GENERATOR
--
-- AUTHOR  :-  Gina Kingston
--
-- DATE    :-  4 Nov 92
--
-- PURPOSE :-  To test the storage implications of assigning a
--             first-class function to variable and returning its
--             value from within the function.
--
-- ** This is a Declarative Ada Dialect (DAD) file **
--
--------------------
with TEXT_IO;
use TEXT_IO;

procedure VARIABLE_RECURSION is

  type FN is function  return INTEGER;
  -- type for the function
```

```
  F : FN;
  -- variable which takes function values

  function FOO : FN;  -- needed so that foo recognises the name foo

  function FOO : FN is
  -- main first-class function which uses the variable F.
    begin
      F := FOO;
      put_line("In Foo");
      return F();
  end;
  function DUMMY: FN is
  -- function for the inital value of the variable
    begin
      return 1;
    end;

begin
  F := DUMMY;
  FOO();
end;
```

B.6 Ada Variable_Recursion Code

```
with LAda_lazy , DAda_box_package ;
with DAda_functions ;
with TEXT_IO ;
use TEXT_IO ;
procedure VARIABLE_RECURSION is
    type DAda_FN_params is record
        null;
    end record ;
    DAda_FN_dummy_params : DAda_FN_params ;
    package DAda_FN_package is new
            DAda_functions ( DAda_FN_params , INTEGER ) ;
    type FN is new DAda_FN_package.funct_type ;
    procedure DAda_FN_complete renames DAda_FN_package.kill_functs ;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER ;
    F : FN ;
    function FOO (
        DAda_params : in DAda_FN_params ) return INTEGER ;
    package DAda_FOO_replacement is new
            DAda_FN_package.funct_generator ( FOO ) ;
    function FOO (
        DAda_params : in DAda_FN_params ) return INTEGER is
    begin
        F := FN(DAda_FOO_replacement.make) ;
        put_line ( "In Foo" ) ;
        return DAda_FN_package.apply ( DAda_FN_dummy_params ,
                DAda_FN_package.funct_type ( F ) ) ;
    end FOO;
    function DUMMY (
        DAda_params : in DAda_FN_params ) return INTEGER is
```

```
    begin
        return 1 ;
    end DUMMY;
    package DAda_DUMMY_replacement is new
            DAda_FN_package.funct_generator ( DUMMY ) ;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER is
    begin
        return DAda_FN_package.apply ( DAda_FN_dummy_params ,
                 DAda_FN_package.funct_type (DAda_fn) ) ;
    end DAda_apply_FN;
begin
    F := FN(DAda_DUMMY_replacement.make) ;
    declare
        DAD_f : INTEGER := FOO ( DAda_FN_dummy_params ) ;
    begin
        null;
    end ;
    DAda_DUMMY_replacement.kill ;
    DAda_FOO_replacement.kill ;
    DAda_FOO_replacement.kill ;
    DAda_FN_package.kill_functs ;
end VARIABLE_RECURSION;
```

B.7 DAD Mutual_Recursion Code

```
--------------------
--
-- PROGRAM :-  MUTUAL_RECURSION
--
-- AUTHOR  :-  Gina Kingston
--
-- DATE    :-  4 Nov 92
--
-- PURPOSE :-  To test the storage implications of mutually recursive
--             first-class functions.
--
-- ** This is a Declarative Ada Dialect (DAD) file **
--
--------------------
with TEXT_IO;
use TEXT_IO;

procedure MUTUAL_RECURSION is

  type FN is function  return INTEGER;
  -- function type for first-class functions


  function DUMMY : FN;
  -- define function for first mutually recursive call
```

```
  function FOO : FN is
  -- mutually recursive with DUMMY
    begin
      put_line("In Foo");
      return DUMMY();  -- mutually recursive call
  end;
  function DUMMY: FN is
  -- mutually recusive with FOO
    begin
      put_line("In Dummy");
      return FOO();  -- mutually recursive call
    end;

begin
  FOO();
end;
```

B.8 Ada Mutual_Recursion Code

```
with LAda_lazy , DAda_box_package ;
with DAda_functions ;
with TEXT_IO ;
use TEXT_IO ;
procedure MUTUAL_RECURSION is
    type DAda_FN_params is record
        null;
    end record ;
    DAda_FN_dummy_params : DAda_FN_params ;
    package DAda_FN_package is new
            DAda_functions ( DAda_FN_params , INTEGER ) ;
    type FN is new DAda_FN_package.funct_type ;
    procedure DAda_FN_complete renames DAda_FN_package.kill_functs ;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER ;
    F : FN ;
    function DUMMY (
        DAda_params : in DAda_FN_params ) return INTEGER ;
    package DAda_DUMMY_replacement is new
            DAda_FN_package.funct_generator ( DUMMY ) ;
    function FOO (
        DAda_params : in DAda_FN_params ) return INTEGER is
    begin
        put_line ( "In Foo" ) ;
        return DUMMY ( DAda_FN_dummy_params ) ;
    end FOO;
    package DAda_FOO_replacement is new
            DAda_FN_package.funct_generator ( FOO ) ;
    function DUMMY (
        DAda_params : in DAda_FN_params ) return INTEGER is
    begin
        put_line ( "In Dummy" ) ;
        return FOO ( DAda_FN_dummy_params ) ;
    end DUMMY;
    function DAda_apply_FN (
        DAda_fn : in FN ) return INTEGER is
```

```
    begin
        return DAda_FN_package.apply ( DAda_FN_dummy_params ,
                DAda_FN_package.funct_type (DAda_fn) ) ;
    end DAda_apply_FN;
begin
    declare
        DAD_f : INTEGER := FOO ( DAda_FN_dummy_params ) ;
    begin
        null;
    end ;
    DAda_DUMMY_replacement.kill ;
    DAda_FOO_replacement.kill ;
    DAda_DUMMY_replacement.kill ;
    DAda_FN_package.kill_functs ;
end MUTUAL_RECURSION;
```

APPENDIX C
Source Code and Output File for the Case Study — TEMPS.

This following sections describe the case study implemented during the evaluation of DAD. The first section contains the original DAD source, the second the generated Ada source and the third is a sample of the type of output produced by the program.

C.1 DAD Temps Code

```
--------------------
--
-- PROGRAM :-  TEMPS
--
-- AUTHOR  :-  Gina Kingston
--
-- DATE    :-  22 Oct 92
--
-- PURPOSE :-  To test how easey it is to write and use of DAD program

--
--              Implements a modified version of a proposed Ada tasking
--              example given by Dr Bo Sanden.
--
-- ** This is a Declarative Ada Dialect (DAD) file **
--
--------------------
with STREAM_PACK;
with CALENDAR; use CALENDAR;
with text_io; use text_io;

procedure TEMPS is

  -- furnace outputs
  type READING is record
    DATE : TIME;
    TEMP : INTEGER;
  end record;
  -- stream for furnace outputs
  package READING_STREAM is new STREAM_PACK(READING);
  function "&" (C : in READING; S : in READING_STREAM.STREAM) return
           READING_STREAM.STREAM renames READING_STREAM."&" ;

  OUTPUT_STREAM : READING_STREAM.STREAM;

  -- stream for furnace inputs

  package INTERVAL_STREAM is new STREAM_PACK(INTEGER);

  function "&" (C : in INTEGER; S : in INTERVAL_STREAM.STREAM) return
           INTERVAL_STREAM.STREAM renames INTERVAL_STREAM."&";
```

```
-- number of furnaces
NUM_FURNACES : constant := 3;
-- used for maniputing the streams for furnace inputs
type INTERVALS is array (1..NUM_FURNACES) of INTERVAL_STREAM.STREAM;
INPUT_STREAMS : INTERVALS;

----------------------------
-- Controls the rate at which the furnaces take samples
----------------------------
   -- **** This function was originally places inside
   -- **** MAINFRAME_INPUT. However the task it generated was
   -- **** similarly nested, but its scope was required to be global
   -- ****

  function INPUT_GENERATION(DUE : TIME; NEW_VALUE : INTEGER) return
           INTERVAL_STREAM.STREAM is
  begin
    delay(DUE - CLOCK);
    return NEW_VALUE & INPUT_GENERATION(DUE + 30.0, NEW_VALUE + 1);
  end INPUT_GENERATION;

function MAINFRAME_INPUT return INTERVALS is
begin
  return ( INPUT_GENERATION(CLOCK + 1.0, 2),
           INPUT_GENERATION(CLOCK + 2.0, 3),
           INPUT_GENERATION(CLOCK + 3.0, 4));
end MAINFRAME_INPUT;

----------------------------
-- Controls the Output to the Screen
----------------------------
procedure MAINFRAME_OUTPUT (OUTPUT_STREAM:
                     in out READING_STREAM.STREAM) is
  procedure OUTPUT(R : READING) is
    YEAR : YEAR_NUMBER;
    MONTH : MONTH_NUMBER;
    DAY : DAY_NUMBER;
    SECONDS : DURATION;
  begin
    SPLIT(R.DATE, YEAR, MONTH, DAY, SECONDS);
    PUT_LINE("RECEIVED "& INTEGER'IMAGE(R.TEMP) & " AT TIME " &
             INTEGER'IMAGE(INTEGER(SECONDS)) & " OF " &
             DAY_NUMBER'IMAGE(DAY) & " / " &
             MONTH_NUMBER'IMAGE(MONTH) & " / " &
             YEAR_NUMBER'IMAGE(YEAR) );
  end OUTPUT;

begin

  READING_STREAM.HEAD_REQUEST(OUTPUT_STREAM);
  loop
    if READING_STREAM.HEAD_AVAILABLE(OUTPUT_STREAM) then
      OUTPUT(READING_STREAM.HEAD(OUTPUT_STREAM));
      OUTPUT_STREAM := READING_STREAM.TAIL(OUTPUT_STREAM);
      READING_STREAM.HEAD_REQUEST(OUTPUT_STREAM);
    else
```

```
      DELAY (0.5);
      -- To enable switching to another function to occur
    end if;
  end loop;
end MAINFRAME_OUTPUT;


----------------------------
-- Merges two streams according to the date part of their elements.
-- Merged streams should already be ordered.
----------------------------

function DATE_BASED_MERGE (S1, S2 : in READING_STREAM.STREAM) return
         READING_STREAM.STREAM is
begin
  READING_STREAM.HEAD_REQUEST(S1);
  READING_STREAM.HEAD_REQUEST(S2);
  if READING_STREAM.HEAD_AVAILABLE(S1) then
    if READING_STREAM.HEAD_AVAILABLE(S2) then
      if READING_STREAM.HEAD (S1).DATE = READING_STREAM.HEAD (S2).
                                         DATE then
        return READING_STREAM.HEAD (S1) & ( READING_STREAM.HEAD (S2)
               & DATE_BASED_MERGE (READING_STREAM.TAIL (S1),
                                   READING_STREAM.TAIL (S2)));
      elsif READING_STREAM.HEAD (S1).DATE < READING_STREAM.HEAD(S2).
                                            DATE then
        return READING_STREAM.HEAD (S1) &
               DATE_BASED_MERGE (READING_STREAM.TAIL (S1), S2);
      else
        return READING_STREAM.HEAD (S2) &
               DATE_BASED_MERGE (S1, READING_STREAM.TAIL (S2));
      end if;
    else
      return READING_STREAM.HEAD (S1) &
             DATE_BASED_MERGE (READING_STREAM.TAIL (S1), S2);
    end if;
  elsif READING_STREAM.HEAD_AVAILABLE(S2) then
    return READING_STREAM.HEAD (S2) &
           DATE_BASED_MERGE (S1, READING_STREAM.TAIL (S2));
  else
    delay 0.5;
    return DATE_BASED_MERGE (S1, S2);
  end if;
end DATE_BASED_MERGE;
----------------------------
-- Controls access to the thermometer
----------------------------


task THERMOMETER is
  entry TEMPERATURE(TEMP: out INTEGER);
end THERMOMETER;

task body THERMOMETER is
  CURRENT_TEMP : INTEGER := 0;
begin
```

```
    loop
      select
        accept TEMPERATURE(TEMP : out INTEGER) do
          TEMP := CURRENT_TEMP;
        end TEMPERATURE;
        CURRENT_TEMP := CURRENT_TEMP + 1;
      or
        terminate;
      end select;
    end loop;
  end THERMOMETER;
  ----------------------------
  -- Procedural call to get the temperature.
  ----------------------------
  -- This is included so that knowledge of the task representation
  -- of the THERMOMETER is not required


  function GET_TEMPERATURE return INTEGER is
    TEMP : INTEGER;
  begin
    THERMOMETER.TEMPERATURE(TEMP);
    return TEMP;
  end GET_TEMPERATURE;

  ----------------------------
  -- Controls the furnaces sampling
  ----------------------------

  function SAMPLE (INT : INTEGER;
                   INPUT_STREAM : INTERVAL_STREAM.STREAM) return
          READING_STREAM.STREAM is
    function SAMPLE_DUE (INT : INTEGER; DUE : TIME;
                         INPUT_STREAM : INTERVAL_STREAM.STREAM) return
            READING_STREAM.STREAM is
    begin
      DELAY(DUE - CLOCK);
      if INTERVAL_STREAM.HEAD_AVAILABLE(INPUT_STREAM) then
        INTERVAL_STREAM.HEAD_REQUEST(INTERVAL_STREAM.TAIL(
                                              INPUT_STREAM));
        return READING_STREAM."&"( (CLOCK, GET_TEMPERATURE),
               SAMPLE_DUE( INTERVAL_STREAM.HEAD(INPUT_STREAM),
                           DURATION(INT * 5) + DUE ,
                           INTERVAL_STREAM.TAIL(INPUT_STREAM)));
        else
          return READING_STREAM."&"( (CLOCK, GET_TEMPERATURE),
                 SAMPLE_DUE(INT, DURATION(INT * 5) + DUE,
                            INPUT_STREAM));
      end if;
    end SAMPLE_DUE;
  begin
    INTERVAL_STREAM.HEAD_REQUEST(INPUT_STREAM);
    return SAMPLE_DUE (INT, CLOCK, INPUT_STREAM);
  end SAMPLE;

-- main program
```

```
begin

  -- 1) set up sampling rate (stream of changes)
  -- 2) set up the stream of results (merged individual streams)
  -- 3) output the results as they arrive at the simulated mainframe

  INPUT_STREAMS := MAINFRAME_INPUT;
  OUTPUT_STREAM := DATE_BASED_MERGE(SAMPLE(1, INPUT_STREAMS(1)),
                     DATE_BASED_MERGE(SAMPLE(2, INPUT_STREAMS(2)),
                                      SAMPLE(3, INPUT_STREAMS(3))));
  MAINFRAME_OUTPUT(OUTPUT_STREAM);

end TEMPS;
```

C.2 Ada Temps Code

```
with LAda_lazy , DAda_box_package ;
with DAda_functions ;
with STREAM_PACK ;
with CALENDAR ;
use CALENDAR ;
with text_io ;
use text_io ;
procedure TEMPS is
    type READING is record
        DATE : TIME ;
        TEMP : INTEGER ;
    end record ;
    package READING_STREAM is new STREAM_PACK ( READING ) ;
    function "&" (
        C : in READING ;
        S : in READING_STREAM.STREAM )
      return READING_STREAM.STREAM renames READING_STREAM . "&" ;
    OUTPUT_STREAM : READING_STREAM.STREAM ;
    package INTERVAL_STREAM is new STREAM_PACK ( INTEGER ) ;
    function "&" (
        C : in INTEGER ;
        S : in INTERVAL_STREAM.STREAM )
      return INTERVAL_STREAM.STREAM renames INTERVAL_STREAM . "&" ;
    NUM_FURNACES : constant := 3 ;
    type INTERVALS is array ( 1 .. NUM_FURNACES )
                  of INTERVAL_STREAM.STREAM ;
    INPUT_STREAMS : INTERVALS ;
    task type LAda_task_INPUT_GENERATION is
        entry INPUT_GENERATION (
            DUE : in TIME ;
            NEW_VALUE : in INTEGER ;
            LAda : in INTERVAL_STREAM.STREAM ) ;
    end LAda_task_INPUT_GENERATION;
    type LAda_task_INPUT_GENERATION_ptr is access
         LAda_task_INPUT_GENERATION ;
    function INPUT_GENERATION (
        DUE : in TIME ;
        NEW_VALUE : in INTEGER ) return INTERVAL_STREAM.STREAM ;
    task type LAda_task_DATE_BASED_MERGE is
```

```
        entry DATE_BASED_MERGE (
            S1 , S2 : in READING_STREAM.STREAM ;
            LAda : in READING_STREAM.STREAM ) ;
    end LAda_task_DATE_BASED_MERGE;
    type LAda_task_DATE_BASED_MERGE_ptr is access
         LAda_task_DATE_BASED_MERGE ;
    function DATE_BASED_MERGE (
        S1 , S2 : in READING_STREAM.STREAM )
      return READING_STREAM.STREAM ;
    task type LAda_task_SAMPLE is
        entry SAMPLE (
            INT : in INTEGER ;
            INPUT_STREAM : in INTERVAL_STREAM.STREAM ;
            LAda : in READING_STREAM.STREAM ) ;
    end LAda_task_SAMPLE;
    type LAda_task_SAMPLE_ptr is access LAda_task_SAMPLE ;
    function SAMPLE (
        INT : in INTEGER ;
        INPUT_STREAM : in INTERVAL_STREAM.STREAM )
      return READING_STREAM.STREAM ;
    task body LAda_task_INPUT_GENERATION is
        DUE : TIME ;
        NEW_VALUE : INTEGER ;
        LAda : INTERVAL_STREAM.STREAM ;
    begin
        accept INPUT_GENERATION (
            DUE : in TIME ;
            NEW_VALUE : in INTEGER ;
            LAda : in INTERVAL_STREAM.STREAM ) do
            LAda_task_INPUT_GENERATION.DUE := DUE ;
            LAda_task_INPUT_GENERATION.NEW_VALUE := NEW_VALUE ;
            LAda_task_INPUT_GENERATION.LAda := LAda ;
        end INPUT_GENERATION ;
        INTERVAL_STREAM.LAda_STREAM_suspend ( LAda ) ;
        declare
        begin
            delay ( DUE - CLOCK ) ;
            INTERVAL_STREAM.LAda_STREAM_transfer_value ( LAda ,
              NEW_VALUE &
              INPUT_GENERATION ( DUE + 30.0 , NEW_VALUE + 1 ) ) ;
            goto LAda_exit ;
            <<LAda_exit>>
            INTERVAL_STREAM.LAda_STREAM_completed ( LAda ) ;
        end ;
    end LAda_task_INPUT_GENERATION;
    function INPUT_GENERATION (
        DUE : in TIME ;
        NEW_VALUE : in INTEGER ) return INTERVAL_STREAM.STREAM is
        LAda : INTERVAL_STREAM.STREAM :=
               INTERVAL_STREAM.LAda_STREAM_create ;
        LAda_active_task : LAda_task_INPUT_GENERATION_ptr :=
                           new LAda_task_INPUT_GENERATION ;
    begin
        LAda_active_task.INPUT_GENERATION ( DUE , NEW_VALUE , LAda ) ;
        return LAda ;
    end INPUT_GENERATION;
```

```
function MAINFRAME_INPUT return INTERVALS is
begin
    return ( INPUT_GENERATION ( CLOCK + 1.0 , 2 ) ,
             INPUT_GENERATION ( CLOCK + 2.0 , 3 ) ,
             INPUT_GENERATION ( CLOCK + 3.0 , 4 ) ) ;
end MAINFRAME_INPUT;
procedure MAINFRAME_OUTPUT (
    OUTPUT_STREAM : in out READING_STREAM.STREAM ) is
    procedure OUTPUT (
        R : in READING ) is
        YEAR : YEAR_NUMBER ;
        MONTH : MONTH_NUMBER ;
        DAY : DAY_NUMBER ;
        SECONDS : DURATION ;
    begin
        SPLIT ( R . DATE , YEAR , MONTH , DAY , SECONDS ) ;
        PUT_LINE ( "RECEIVED " & INTEGER ' IMAGE ( R . TEMP ) &
                   " AT TIME " &
                   INTEGER ' IMAGE ( INTEGER ( SECONDS ) ) &
                   " OF " & DAY_NUMBER ' IMAGE ( DAY ) &
                   " / " & MONTH_NUMBER ' IMAGE ( MONTH ) &
                   " / " & YEAR_NUMBER ' IMAGE ( YEAR ) ) ;
    end OUTPUT;
begin
    READING_STREAM . HEAD_REQUEST ( OUTPUT_STREAM ) ;
    loop
        if READING_STREAM . HEAD_AVAILABLE ( OUTPUT_STREAM ) then
            OUTPUT ( READING_STREAM . HEAD ( OUTPUT_STREAM ) ) ;
            OUTPUT_STREAM := READING_STREAM . TAIL
                                ( OUTPUT_STREAM ) ;
            READING_STREAM . HEAD_REQUEST ( OUTPUT_STREAM ) ;
        else
            delay ( 0.5 ) ;
        end if;
    end loop ;
end MAINFRAME_OUTPUT;
task body LAda_task_DATE_BASED_MERGE is
    S1 , S2 : READING_STREAM.STREAM ;
    LAda : READING_STREAM.STREAM ;
begin
    accept DATE_BASED_MERGE (
        S1 , S2 : in READING_STREAM.STREAM ;
        LAda : in READING_STREAM.STREAM ) do
        LAda_task_DATE_BASED_MERGE.S1 := S1 ;
        LAda_task_DATE_BASED_MERGE.S2 := S2 ;
        LAda_task_DATE_BASED_MERGE.LAda := LAda ;
    end DATE_BASED_MERGE ;
    READING_STREAM.LAda_STREAM_suspend ( LAda ) ;
    declare
    begin
        READING_STREAM . HEAD_REQUEST ( S1 ) ;
        READING_STREAM . HEAD_REQUEST ( S2 ) ;
        if READING_STREAM . HEAD_AVAILABLE ( S1 ) then
            if READING_STREAM . HEAD_AVAILABLE ( S2 ) then
                if READING_STREAM . HEAD ( S1 ) . DATE =
                   READING_STREAM . HEAD ( S2 ) . DATE then
```

```
                    READING_STREAM.LAda_STREAM_transfer_value
                     ( LAda , READING_STREAM . HEAD ( S1 ) &
                       ( READING_STREAM . HEAD ( S2 ) &
                         DATE_BASED_MERGE (
                           READING_STREAM . TAIL ( S1 ) ,
                           READING_STREAM . TAIL ( S2 ) ) ) ) ;
                    goto LAda_exit ;
                elsif READING_STREAM . HEAD ( S1 ) . DATE <
                      READING_STREAM . HEAD ( S2 ) . DATE then
                    READING_STREAM.LAda_STREAM_transfer_value (
                      LAda , READING_STREAM . HEAD ( S1 ) &
                      DATE_BASED_MERGE (
                        READING_STREAM . TAIL ( S1 ) , S2 ) ) ;
                    goto LAda_exit ;
                else
                    READING_STREAM.LAda_STREAM_transfer_value
                      ( LAda , READING_STREAM . HEAD ( S2 ) &
                        DATE_BASED_MERGE ( S1 ,
                          READING_STREAM . TAIL ( S2 ) ) ) ;
                    goto LAda_exit ;
                end if;
            else
                READING_STREAM.LAda_STREAM_transfer_value ( LAda ,
                  READING_STREAM . HEAD ( S1 ) &
                  DATE_BASED_MERGE (
                    READING_STREAM . TAIL ( S1 ) , S2 ) ) ;
                goto LAda_exit ;
            end if;
        elsif READING_STREAM . HEAD_AVAILABLE ( S2 ) then
            READING_STREAM.LAda_STREAM_transfer_value ( LAda ,
              READING_STREAM . HEAD ( S2 ) &
              DATE_BASED_MERGE ( S1 ,
              READING_STREAM . TAIL ( S2 ) ) ) ;
            goto LAda_exit ;
        else
            delay 0.5 ;
            READING_STREAM.LAda_STREAM_transfer_value ( LAda ,
              DATE_BASED_MERGE ( S1 , S2 ) ) ;
            goto LAda_exit ;
        end if;
        <<LAda_exit>>
        READING_STREAM.LAda_STREAM_completed ( LAda ) ;
    end ;
end LAda_task_DATE_BASED_MERGE;
function DATE_BASED_MERGE (
    S1 , S2 : in READING_STREAM.STREAM )
  return READING_STREAM.STREAM is
    LAda : READING_STREAM.STREAM :=
           READING_STREAM.LAda_STREAM_create ;
    LAda_active_task : LAda_task_DATE_BASED_MERGE_ptr :=
                       new LAda_task_DATE_BASED_MERGE ;
begin
    LAda_active_task.DATE_BASED_MERGE ( S1 , S2 , LAda ) ;
    return LAda ;
end DATE_BASED_MERGE;
task THERMOMETER is
```

```
    entry TEMPERATURE (
        TEMP : out INTEGER ) ;
end THERMOMETER;
task body THERMOMETER is
    CURRENT_TEMP : INTEGER := 0 ;
begin
    loop
        select
            accept TEMPERATURE (
                TEMP : out INTEGER ) do
                TEMP := CURRENT_TEMP ;
            end TEMPERATURE ;
            CURRENT_TEMP := CURRENT_TEMP + 1 ;
        or
            terminate;
        end select;
    end loop ;
end THERMOMETER;
function GET_TEMPERATURE return INTEGER is
    TEMP : INTEGER ;
begin
    THERMOMETER . TEMPERATURE ( TEMP ) ;
    return TEMP ;
end GET_TEMPERATURE;
task body LAda_task_SAMPLE is
    INT : INTEGER ;
    INPUT_STREAM : INTERVAL_STREAM.STREAM ;
    LAda : READING_STREAM.STREAM ;
begin
    accept SAMPLE (
        INT : in INTEGER ;
        INPUT_STREAM : in INTERVAL_STREAM.STREAM ;
        LAda : in READING_STREAM.STREAM ) do
        LAda_task_SAMPLE.INT := INT ;
        LAda_task_SAMPLE.INPUT_STREAM := INPUT_STREAM ;
        LAda_task_SAMPLE.LAda := LAda ;
    end SAMPLE ;
    READING_STREAM.LAda_STREAM_suspend ( LAda ) ;
    declare
        task type LAda_task_SAMPLE_DUE is
            entry SAMPLE_DUE (
                INT : in INTEGER ;
                DUE : in TIME ;
                INPUT_STREAM : in INTERVAL_STREAM.STREAM ;
                LAda : in READING_STREAM.STREAM ) ;
        end LAda_task_SAMPLE_DUE;
        type LAda_task_SAMPLE_DUE_ptr is access
             LAda_task_SAMPLE_DUE ;
        function SAMPLE_DUE (
            INT : in INTEGER ;
            DUE : in TIME ;
            INPUT_STREAM : in INTERVAL_STREAM.STREAM )
          return READING_STREAM.STREAM ;
        task body LAda_task_SAMPLE_DUE is
            INT : INTEGER ;
            DUE : TIME ;
```

```
        INPUT_STREAM : INTERVAL_STREAM.STREAM ;
        LAda : READING_STREAM.STREAM ;
    begin
        accept SAMPLE_DUE (
            INT : in INTEGER ;
            DUE : in TIME ;
            INPUT_STREAM : in INTERVAL_STREAM.STREAM ;
            LAda : in READING_STREAM.STREAM ) do
            LAda_task_SAMPLE_DUE.INT := INT ;
            LAda_task_SAMPLE_DUE.DUE := DUE ;
            LAda_task_SAMPLE_DUE.INPUT_STREAM :=
                                    INPUT_STREAM ;
            LAda_task_SAMPLE_DUE.LAda := LAda ;
        end SAMPLE_DUE ;
        READING_STREAM.LAda_STREAM_suspend ( LAda ) ;
        declare
        begin
            delay ( DUE - CLOCK ) ;
            if INTERVAL_STREAM . HEAD_AVAILABLE
                                ( INPUT_STREAM ) then
                INTERVAL_STREAM . HEAD_REQUEST
                 ( INTERVAL_STREAM . TAIL ( INPUT_STREAM ) ) ;
                READING_STREAM.LAda_STREAM_transfer_value
                 ( LAda , READING_STREAM . "&" (
                    ( CLOCK , GET_TEMPERATURE ) ,
                    SAMPLE_DUE (
                     INTERVAL_STREAM . HEAD ( INPUT_STREAM ) ,
                     DURATION ( INT * 5 ) + DUE ,
                     INTERVAL_STREAM . TAIL ( INPUT_STREAM
                ) ) ) ) ;
                goto LAda_exit ;
            else
                READING_STREAM.LAda_STREAM_transfer_value (
                  LAda , READING_STREAM . "&" (
                    ( CLOCK , GET_TEMPERATURE ) ,
                    SAMPLE_DUE ( INT ,
                                 DURATION ( INT * 5 ) + DUE ,
                                 INPUT_STREAM ) ) ) ;
                goto LAda_exit ;
            end if;
            <<LAda_exit>>
            READING_STREAM.LAda_STREAM_completed ( LAda ) ;
        end ;
    end LAda_task_SAMPLE_DUE;
    function SAMPLE_DUE (
        INT : in INTEGER ;
        DUE : in TIME ;
        INPUT_STREAM : in INTERVAL_STREAM.STREAM )
      return READING_STREAM.STREAM is
        LAda : READING_STREAM.STREAM :=
               READING_STREAM.LAda_STREAM_create ;
        LAda_active_task : LAda_task_SAMPLE_DUE_ptr :=
                           new LAda_task_SAMPLE_DUE ;
    begin
        LAda_active_task.SAMPLE_DUE
          ( INT , DUE , INPUT_STREAM , LAda ) ;
```

```
            return LAda ;
        end SAMPLE_DUE;
    begin
        INTERVAL_STREAM . HEAD_REQUEST ( INPUT_STREAM ) ;
        READING_STREAM.LAda_STREAM_transfer_value ( LAda ,
          SAMPLE_DUE ( INT , CLOCK , INPUT_STREAM ) ) ;
        goto LAda_exit ;
        <<LAda_exit>>
        READING_STREAM.LAda_STREAM_completed ( LAda ) ;
    end ;
end LAda_task_SAMPLE;
function SAMPLE (
    INT : in INTEGER ;
    INPUT_STREAM : in INTERVAL_STREAM.STREAM )
  return READING_STREAM.STREAM is
    LAda : READING_STREAM.STREAM :=
           READING_STREAM.LAda_STREAM_create ;
    LAda_active_task : LAda_task_SAMPLE_ptr :=
                       new LAda_task_SAMPLE ;
begin
    LAda_active_task.SAMPLE ( INT , INPUT_STREAM , LAda ) ;
    return LAda ;
end SAMPLE;
begin
    INPUT_STREAMS := MAINFRAME_INPUT ;
    OUTPUT_STREAM :=
      DATE_BASED_MERGE ( SAMPLE ( 1 , INPUT_STREAMS ( 1 ) ) ,
        DATE_BASED_MERGE ( SAMPLE ( 2 , INPUT_STREAMS ( 2 ) ) ,
                           SAMPLE ( 3 , INPUT_STREAMS ( 3 ) ) ) ) ;
    MAINFRAME_OUTPUT ( OUTPUT_STREAM ) ;
end TEMPS;
```

C.3 Temps Output

```
RECEIVED  0 AT TIME  43282 OF  4 /  11 /  1992
RECEIVED  1 AT TIME  43282 OF  4 /  11 /  1992
RECEIVED  2 AT TIME  43282 OF  4 /  11 /  1992
RECEIVED  3 AT TIME  43288 OF  4 /  11 /  1992
RECEIVED  4 AT TIME  43293 OF  4 /  11 /  1992
RECEIVED  5 AT TIME  43294 OF  4 /  11 /  1992
RECEIVED  6 AT TIME  43299 OF  4 /  11 /  1992
RECEIVED  7 AT TIME  43303 OF  4 /  11 /  1992
RECEIVED  8 AT TIME  43304 OF  4 /  11 /  1992
RECEIVED  9 AT TIME  43313 OF  4 /  11 /  1992
RECEIVED  10 AT TIME  43314 OF  4 /  11 /  1992
RECEIVED  11 AT TIME  43319 OF  4 /  11 /  1992
RECEIVED  12 AT TIME  43324 OF  4 /  11 /  1992
RECEIVED  13 AT TIME  43334 OF  4 /  11 /  1992
RECEIVED  14 AT TIME  43334 OF  4 /  11 /  1992
RECEIVED  15 AT TIME  43335 OF  4 /  11 /  1992
RECEIVED  16 AT TIME  43344 OF  4 /  11 /  1992
RECEIVED  17 AT TIME  43349 OF  4 /  11 /  1992
RECEIVED  18 AT TIME  43354 OF  4 /  11 /  1992
RECEIVED  19 AT TIME  43355 OF  4 /  11 /  1992
RECEIVED  20 AT TIME  43363 OF  4 /  11 /  1992
```

RECEIVED 21 AT TIME 43364 OF 4 / 11 / 1992
RECEIVED 22 AT TIME 43374 OF 4 / 11 / 1992
RECEIVED 23 AT TIME 43375 OF 4 / 11 / 1992
RECEIVED 24 AT TIME 43379 OF 4 / 11 / 1992
RECEIVED 25 AT TIME 43384 OF 4 / 11 / 1992
RECEIVED 26 AT TIME 43393 OF 4 / 11 / 1992
RECEIVED 27 AT TIME 43394 OF 4 / 11 / 1992
RECEIVED 28 AT TIME 43395 OF 4 / 11 / 1992
RECEIVED 29 AT TIME 43404 OF 4 / 11 / 1992
RECEIVED 30 AT TIME 43409 OF 4 / 11 / 1992
RECEIVED 31 AT TIME 43414 OF 4 / 11 / 1992
RECEIVED 32 AT TIME 43415 OF 4 / 11 / 1992
RECEIVED 33 AT TIME 43424 OF 4 / 11 / 1992
RECEIVED 34 AT TIME 43424 OF 4 / 11 / 1992
RECEIVED 35 AT TIME 43433 OF 4 / 11 / 1992
RECEIVED 36 AT TIME 43435 OF 4 / 11 / 1992
RECEIVED 37 AT TIME 43439 OF 4 / 11 / 1992
RECEIVED 38 AT TIME 43444 OF 4 / 11 / 1992
RECEIVED 39 AT TIME 43453 OF 4 / 11 / 1992
RECEIVED 40 AT TIME 43454 OF 4 / 11 / 1992
RECEIVED 41 AT TIME 43455 OF 4 / 11 / 1992
RECEIVED 42 AT TIME 43464 OF 4 / 11 / 1992
RECEIVED 43 AT TIME 43469 OF 4 / 11 / 1992
RECEIVED 44 AT TIME 43474 OF 4 / 11 / 1992
RECEIVED 45 AT TIME 43475 OF 4 / 11 / 1992
RECEIVED 46 AT TIME 43483 OF 4 / 11 / 1992
RECEIVED 47 AT TIME 43484 OF 4 / 11 / 1992
RECEIVED 48 AT TIME 43493 OF 4 / 11 / 1992
RECEIVED 49 AT TIME 43495 OF 4 / 11 / 1992
RECEIVED 50 AT TIME 43499 OF 4 / 11 / 1992
RECEIVED 51 AT TIME 43504 OF 4 / 11 / 1992
RECEIVED 52 AT TIME 43514 OF 4 / 11 / 1992
RECEIVED 53 AT TIME 43514 OF 4 / 11 / 1992
RECEIVED 54 AT TIME 43515 OF 4 / 11 / 1992

APPENDIX D
Problems Encountered while Compiling and Executing DAD Programs

There was another discovery found when writing the VARIABLE_RECURSION.d program. It is very easy to work around and may or may not be considered to be erroneous. The .d file contained such a minor differences to the final VARIABLE_RECURSION that it was not considered worthwhile to include it separately. In the first-class function *foo* a variable is assigned to take the value *foo*. The translation of this into Ada is incorrect when the function has not be specified separately first. However in the example given the program is analogous to SIMPLE_RECURSION. The only difference being the use of the variable.

PROGRAM	COMPILER/LINKER			PROBLEMS
	Telesoft	VAX	Verdix	
DAD standard files	OK	1	OK	1 - lada_lazy.a
addn	OK	OK	-	
ham	2	2	-	2 - ham.ada
mapfib	2	2	-	2 - mapfib.ada
mapprime	2	2	-	2 - mapprime.ada
primes	OK	OK	-	
sfu	3	OK	OK	3
setst	2	2	-	2 - sets.ada
yhand	2	2	-	2 - yhand.ada
ystub	4	4	-	4
ytest	4	4	-	2 - ytest.ada 4
jstup1	2	2	-	2 - jstup1.ada 5
alt_bit	2,3	2	-	2 - alt_bit.ada 3
alt_bit_verb	2,3	2	-	2 - alt_bit_verb.ada 3
dms	6	6	7	6, 7
cruise_control	3,8	8	7	3, 7, 8
stream_test	-	-	-	9 - stream_test.d
trial	OK	-	-	
trial_2	OK	-	-	
trial_3	OK	-	-	
trial_4	OK	-	-	
temps_1	OK	-	-	
temps_2	OK	-	-	

Table 1 DAD Programs and the Problems Encountered While Trying to Execute Them (Continued . . .)

PROGRAM	COMPILER/LINKER			PROBLEMS
	Telesoft	VAX	Verdix	
temps_3	-	-	-	10 - temps_3.d
temps_4	OK	OK	-	

Table 1 DAD Programs and the Problems Encountered While Trying to Execute Them

#	PROBLEM	COMPILER	FIXABLE/ UNFIXED
1.	**Tasking Priority Range** Pragma Priority was used with values 5, 10 and 20. Vax only has priority values 0..15 (Default 7) so the values had to be changed to lie within this range. Note that their order relative to each other was maintained, but not with respect to the default value. Telesoft has priorities in the range 0..63 (Default 31). Verdix has priorites in the range 0..99 (Default 0).	VAX	FIXABLE
2.	**Number of Characters per Line of Code** The DAD preprocessor generates Ada code with a maximum of 250 characters per line. The Telesoft compiler requires line lengths to be a maximum of 200 characters long. The VAX requires lines to be at most 120 characters long. Hence some of the generated files needed to be modified before they could be compiled. Verdix has a maximum of 499 characters per line.	Telesoft VAX	FIXABLE
3.	**PROGRAM_ERROR** A Program_Error occurred during Execution. For one program it appears as if a call to TEXT_IO.OPEN caused the problem. The others could not be debugged as the main program consisted of statements which were calls to instantiations of generics. The Telesoft compiler has a known bug whereby generics and their instantiations cannot be debugged.	Telesoft	UNFIXED
4.	**Termination** Two of the programs which should have terminated did not. As *NAME*.kill is called when a task is no longer required, it appears as if at least one task was missed.	N/A	UNFIXED
5.	**READ_ME** The compilation order in the READ_ME file was wrong. One of the standard DAD libraries is overwritten by the misplaced file.	N/A	FIXABLE

Table 2 Problems Encountered While Trying To Execute DAD Programs (Continued . . .)

6.	**Generic Nesting** In some cases the compiler had difficulty due to the depth of nesting of the generics. This was fixed by directly instantiating one or more levels of the generics. The Telesoft error message gave no clues as to the cause of failure. *>>> SEMANTIC: Exception in Middle Pass* *>>> Internal Error: Operation Aborted.*	Telesoft VAX	FIXABLE
7.	**Load Error** The error *ld : VERDIX_HOME/standard/-objects/link_block_b01 : internal error* occurred during linking. The cause of this error could not be determined.	Verdix	UNFIXED
8.	**Maths Library** Telesoft, VAX and Verdix all have differently named maths libraries. Telesoft's is called *generic_elementary_functions*, VAX's is called *math_lib* and Verdix's is called *math*. This needed to be changed to the correct library from *math*.	Telesoft VAX	FIXABLE
9.	**GOTO in Exception Handler** A GOTO was placed in the exception handler by the DAD preprocessor. This occured for a first-class function and would probably occur for any function which must be translated to a task by the preprocessor.	N/A	N/A
10.	***Scope of TASK Incorrect*** When a function which returns a lazy type is nested the task associated with it in the generated Ada code may have the wrong scope. This occurs if the evaluation of the lazy datastructure is only required outside of the scope of its containing function.	N/A	N/A

Table 2 Problems Encountered While Trying To Execute DAD Programs

APPENDIX E
DAD Software

This appendix describes the structure of the DAD directory as it was delivered, including descriptions of the DAD commands, library packages and sample programs.

E.1 Directory Structure

Table 3 describes the main directories of the DAD directory.

DIRECTORY	CONTAINS
`bin`	The scripts and binaries for creating and removing DAD libraries and preprocessing DAD files.
`demo`	The sample systems delivered. The subdirectories are numbered according to the document in [2] where the systems are discussed.
`lib`	The DAD runtime system (written in standard Ada) and DAD library packages (written in DAD).
`src`	The source files for the DAD preprocessor. `DAD_COM` is a shell script for building the system.

Table 3 DAD Directory Structure

E.2 DAD Preprocessor

The files in the `bin` directory are described in Table 4, the files in the `demo` directory are described in the section on the demonstrations and the files in the `lib` directory are described in Table 5.

COMMAND	PURPOSE	EXAMPLE
`dad`	Preprocesses DAD files (suffix `.d`) to obtain Ada files (suffix `.ada`) as described in the manual page at the end of [5]. (The `-i` option for generating illuminated listings and `-a` option for invoking the compiler have not been implemented for the Telesoft version of the preprocessor. This binary is compiled for the Sun 4 architecture.)	`dad addn.d`
`d.all`	Preprocesses a DAD file and then compiles and links the resulting Ada file. NOTE: 1) The DAD file must contain a main unit with the same name as the file. 2) The current directory must be setup for DAD.	`d.all addn`
`d.mklib`	Sets up the current directory for DAD. (See below).	
`d.rmlib`	Deletes all files generated by DAD and Ada commands. (See below).	

Table 4 DAD `bin` Directory

The `d.mklib` and `d.rmlib` scripts do not work correctly for Telesoft. To perform the equivalent of `d.mklib`, do the following:

1. Ensure that your environment variables are correctly set for the Telesoft tools. `$TELESOFT` (not `$TELESOFT/bin`) should be in your path.

2. Create a Telesoft library (e.g. `samplib`) and `liblst.alb` file.
3. Make the `.dada`, `.dada_syms` and `.debug_table` subdirectories.
4. Run the `ln`, `dad`, and `ada` commands as in `d.mklib`.

To perform the equivalent of `d.rmlib`, do the following:

```
rm -rf .dada .dada_syms .debug_table *.ada
rm -rf samplib.obj samplib.sub
```

where `samplib` is the name of the Ada library created above.

FILE	PROVIDES
stream_pack.d	A generic lazy stream package.
a_strings.a	A package for variable length strings.
a_files.d	A package which allows standard Ada files to be passed as parameters to DAD functions and procedures.

Table 5 DAD 'lib' Directory

E.3 Demonstration Programs

The DAD source code for several demonstration programs was delivered along with the DAD preprocessor. These programs reside in the demo subdirectory of the DAD directory.

#	PROGRAM	DEMO DIRECTORY	ACTIONS
1	addn	6	An example of currying using functions which add 'n' to a number.
2	ham	6	Generates the Hamming number sequence.
3	mapfib	6	Returns the Fibonacci numbers with prime indices.
4	mapprime	6	Generates the squares of prime numbers.
5	primes	6	An implementation of the Sieve of Eratosthenes.
6	sfu	6	An example of using streams for sequential file updates.
7	setst	7	An implemenation of sets using their characteristic functions.
8	yhand	7	λ-expressions using hand generated laziness.
9	ystub	7	λ-expressions
10	ytest	7	λ-expressions using first-class functions.
11	jstup1	8	A solution to the Jobshop problem using streams.
12	alt_bit	9	Alternating Bit Protocol implementation.
13	alt_bit_verb	9	As above, with some generics expanded (to get around Verdix Ada restrictions)

Table 6 Demonstration Programs (Continued . . .)

#	PROGRAM	DEMO DIRECTORY	ACTIONS
14	dms	10	An implementation of a Dependency Management System.
15	cruise_control	11	An implementation of a Cruise Control for a car, with a simulator.

Table 6 Demonstration Programs

The programs are grouped into further subdirectories according to where they are documented. The number of the subdirectory corresponds to the number of the document describing its contents in [2]. The names of the demonstrations, their locations and, their intended function are given in Table 6.

DISTRIBUTION

	No. of Copies
Defence Science and Technology Organisation	
Chief Defence Scientist)	
Central Office Executive)	1 shared copy
Counsellor, Defence Science, London	Cont Sht
Counsellor, Defence Science, Washington	Cont Sht
Scientific Adviser POLCOM	1 copy
Senior Defence Scientific Adviser	1 copy
Assistant Secretary Scientific Analysis	1 copy
Navy Office	
Navy Scientific Adviser	1 copy
Air Office	
Air Force Scientific Adviser	1 copy
Army Office	
Scientific Adviser, Army	1 copy
Electronics Research Laboratory	
Director	1 copy
Chief, Information Technology Division	1 copy
Chief, Electronic Warfare Division	Cont Sht
Chief, Communications Division	Cont Sht
Chief, Guided Weapons Division	Cont Sht
Research Leader Military Computing Systems	1 copy
Research Leader Command, Control & Intelligence Systems Branch	1 copy
Head Software Engineering Group	1 copy
G. Kingston (Author)	3 copies
S. Crawley (Author)	1 copy
Head, Program and Executive Support	1 copy
Manager Human Computer Interaction Laboratory	1 copy
Head, Command Support Systems Group	1 copy
Head, Intelligence Systems Group	1 copy
Head, Systems Simulation and Assessment Group	1 copy
Head, Exercise Analysis Group	1 copy
Head, C3I Systems Engineering Group	1 copy
Head, Computer Systems Architecture Group	1 copy
Head, Trusted Computer Systems Group	1 copy
Head, Information Management Group	1 copy
Head, Information Acquisition & Processing Group	1 copy
Publications & Publicity Officer, Information Technology Division	1 copy
P. Bailes (University of Queensland)	2 copies

Libraries and Information Services

Australian Government Publishing Service	1 copy
Defence Central Library, Technical Reports Centre	1 copy
Manager, Document Exchange Centre, (for retention)	1 copy
National Technical Information Service, United States	2 copies
Defence Research Information Centre, United Kingdom	2 copies
Director Scientific Information Services, Canada	1 copy
Ministry of Defence, New Zealand	1 copy
National Library of Australia	1 copy
Defence Science and Technology Organisation Salisbury, Research Library	2 copies
Library Defence Signals Directorate	1 copy
British Library Document Supply Centre	1 copy

Spares

Defence Science and Technology Organisation Salisbury, Research Library	6 copies

Department of Defence

DOCUMENT CONTROL DATA SHEET

1. Page Classification	Unclassified
2. Privacy Marking/Caveat (of document)	

3a. AR Number	3b. Laboratory Number	3c. Type of Report	4. Task Number
AR-008-466	ERL-0715-RN	Research Note	

5. Document Date	6. Cost Code	8. No. of Pages	62
JANUARY 1994		9. No. of Refs.	13

10. Title

AN INDEPENDENT EVALUATION OF THE DECLARATIVE ADA DIALECT

7. Security Classification

*			
U	U	U	
Document	Title	Abstract	

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L) in document box.

11. Author(s)

Gina Kingston and Stephen Crawley

12. Downgrading/Delimiting Instructions

13a. Corporate Author and Address

Electronics Research Laboratory
PO Box 1500, Salisbury SA 5108

13b. Task Sponsor

14. Officer/Position responsible for

Security:.....

Downgrading:.....

Approval forRelease:...DERL.....

15. Secondary Release Statement of this Document

APPROVED FOR PUBLIC RELEASE

16a. Deliberate Announcement

No Limitation

16b. Casual Announcement (for citation in other documents)

[X] No Limitation [ ] Ref. by Author , Doc No. and date only.

17. DEFTEST Descriptors

Ada (programming language)
Functional programming
Declarative Ada Dialect *
Prototypes

18. DISCAT Subject Codes

1205

19. Abstract

This paper provides an independent evaluation of the Declarative Ada Dialect (DAD), which allows functional-style programming in Ada and was developed by the University of Queensland under a Research Agreement with DSTO. It describes the use of DAD and discusses its benefits and limitations.